Software Engineering - Einführung
(SE I bzw. Analyse & Design)



Software Engineers' Death March Project

Prof. Dr. Andy Schürr
Fachgebiet Echtzeitsysteme
FB 18 & FB 20

Technische Universität Darmstadt,
Merckstr. 25, D-64283 Darmstadt

Andy.Schuerr@es.tu-darmstadt.de

Tel.: 06151 / 16-6940
Raum: S 306 / 313

WWW-Seite der Vorlesung:

http://www.es.tu-darmstadt.de/lehre/se-i-v/

Bildquelle:

Jules Verne: The Great Explorers of the XIXth Century,
New York: Charles Scribner's Sons (1912)



Inhaltsverzeichnis der Vorlesung:



Zielsetzungen der Vorlesung:

- den Zuhörer von der Notwendigkeit der Software-Technik überzeugen
- Basis für fortführende Software-Engineering-Lehrveranstaltungen bilden (und Interesse an dieser Thematik wecken)
- solides Handwerkszeug für Durchführung von Bachelor-Arbeiten etc. vermitteln
- dabei Konzentration auf objektorientierte Standardmodellierungssprache UML
- erste Erfahrungen mit „Computer-Aided-Software-Engineering“-Werkzeugen
- die Erhebung, Analyse und Dokumentation von Anforderungen lernen mit halbformalen Methoden (mit grafischen Notationen)
- das Design von Software-Architekturen und Umsetzung in Code üben
- Einblick in das Design von Datenbanken erhalten
- formale(re) Grundlagen von UML-Teilen durch Petri-Netz-Exkurs



Übungen:

- Organisation als Teams von jeweils etwa 3 bis 4 Studenten
- es gibt korrigierte Hausaufgaben/Übungsaufgaben
- in Präsenzübungen werden Aufgaben anhand von Musterlösungen besprochen
- Einsatz eines CASE-Tools in Rechnerpools oder zum Selberinstallieren

Bonussystem:

- ⇨ maximaler Bonus von 0,4 auf Klausurnote (falls bestanden)
- ⇨ jedes Übungsblatt mit mindestens 50% der Punkte ergibt 0,05 Bonus
- ⇨ maximal 8 von insgesamt 11 Übungsblätter werden also berücksichtigt

Betreuer:

- ☞ **Erhan Leblebici** (erhan.leblibici@es.tu-darmstadt.de)
- ☞ **Gergely Varró** (gergely.varro@es.tu-darmstadt.de)



Organisation der Lehrveranstaltung:

- **Termine** (für Spätaufsteher)
  - Mittwoch: 16:15 Uhr bis 17:45 in S1|01|A03 (Vorlesung ab 15.10.14)
  - Donnerstag: 16:15 bis 17:00 Uhr in S1|01|A03 (Vorlesung ab 16.10.14)
    Donnerstag: 17:05 bis 17:50 Uhr in S1|01|A03 (Übung ab 23.10.14)
- **Klausur**
  - Klausur am Ende der Vorlesungszeit: 25.02.2015 ab 8:00 Uhr
  - alte Klausuraufgaben werden online gestellt (mit Musterlösung)



Quellenverweis und Danksagung:

Für die Erstellung der ersten Version dieser Vorlesung stand mir dankenswerter Weise das Folienmaterial der Software-Engineering-Vorlesung von

Prof. Dr. Gregor Engels
Universität Paderborn

zur Verfügung. Bei den Überarbeitungen dieser Vorlesung wurden die übernommenen Teile aber inzwischen stark überarbeitet, so dass davon auszugehen ist, dass alle Fehler bezüglich Inhalt, Layout und Umgang mit der deutschen Sprache allein dem Dozenten der Lehrveranstaltung zuzuschreiben sind.

Des weiteren habe ich Anregungen und Bilder aus dem Vorlesungsmaterial von

Dr. Michael Eichberg
TU Darmstadt

übernommen (Vorlesungsunterlagen von EiSE WS 2007/2008).



Wichtige Literaturquellen:

[Ba96] H. Balzert: *Lehrbuch der Software-Technik (Band 1): Software-Entwicklung*, Spektrum Akademischer Verlag (2000), 2-te Auflage, 1136 Seiten

Sehr umfangreiches und gut lesbares Nachschlagewerk mit CD. Auf der CD findet man Werkzeuge, Videos, Übungsaufgaben mit Lösungen (aber nicht unsere), …

[Ba99] H. Balzert: *Lehrbuch der Objektmodellierung - Analyse und Entwurf*, Spektrum Akademischer Verlag (1999), 573 Seiten

Im gleichen Stil wie [Ba96] geschriebenes Buch, das eine gute Ergänzung zu Kapitel 5 und 6 der Vorlesung darstellt. Wieder ist eine CD mit Werkzeugen, … beigelegt.

[So07] I. Sommerville: *Software Engineering*, Addison-Wesley - Pearson Studium, 6. Auflage (2007), 875 Seiten

In Deutsch übersetztes Standardlehrbuch, das sehr umfassend alle wichtigen Themen der Software-Technik knapp behandelt. Empfehlenswert!

[St05] H. Störrle: *UML2 für Studenten*, Pearson Studium (2005), 320 Seiten

Leider bislang nur in Deutsch verfügbare Einführung in UML2, die Standard-Softwaremodellierungssprache (Version 2); beschränkt sich auf wichtige Sprachelemente und verwendet ein durchgängiges Beispiel (ähnlich wie diese Lehrveranstaltung). Das Buch ist ohne Vorkenntnisse gut zu verstehen.

[GH95] E. Gamma, R. Helm, und R. E. Johnson: Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley (1995), 385 Seiten

Der Klassiker zum Thema wiederverwendbarer Software-Entwurfsmuster



Ergänzende Literaturquellen:

[Ka98] H. Balzert: *Lehrbuch der Software-Technik (Band 2): Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum Akademischer Verlag (1998), 769 Seiten

Hier findet man fast alles über das Gebiet Software-Technik, was nicht bereits in [Ba96] abgehandelt wurde. Wieder ist eine CD mit den entsprechenden Werkzeugen, Übungsaufgaben, Musterlösungen, … beigelegt.

[Be00] R.V. Beizer: *Testing Object-Oriented Systems - Models, Patterns, Tools*, Addison-Wesley (2000), 1191 Seiten

Die zur Zeit umfassendste Quelle zum angesprochenen Thema. Gut geschrieben und umbedingt empfehlenswert, falls man die Thematik „Testen“ vertiefen will.

[BRJ99] G. Booch, J. Rumbaugh, I. Jacobson: *Das UML Benutzerhandbuch*, Addison Wesley (2006), 543 Seiten

Das “offizielle” UML-Einführungsbuch der drei UML-Väter. Es beschreibt die verschiedenen Diagrammarten aus Benutzersicht, allerdings ohne größere zusammenhängende Beispiele zu verwenden. Allgemeine Software-Techniken bleiben aussen vor. Nicht meine Empfehlung!

[GJM91] C. Ghezzi, M. Jazayeri, D. Mandrioli: *Fundamentals of Software Engineering*, Prentice Hall (1991), 573 Seiten

Klassisches Software-Technikbuch mit stärker lehrbuchartigem Charakter (im Vergleich zu [Ba96]). Naturgemäß leider nicht mehr auf der Höhe der Zeit, trotzdem aber lesenswert.



[HK05] M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger: *UML@Work*, 3-te Auflage, dpunkt.verlag (2005), 422 Seiten

Sehr schön geschriebenes UML-Buch mit durchgängigem Beispiel, das auch auf die „düsteren" Seiten der UML eingeht und umstrittene Sprachkonstrukte ausführlich diskutiert.

[JB99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*, Addison Wesley (1999)

Präsentation eines auf UML zugeschnittenen Vorgehensmodells der Software-Entwicklung.

[Ka98] B. Kahlbrandt: *Software-Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language*, Springer Verlag (1998), 504 Seiten

Mit das Buch, das vom Inhalt her dieser Vorlesung am nächsten steht. Der Schwerpunkt liegt auf dem Einsatz von UML; allgemeinere Themen wie Kostenschätzung und Testen werden dafür ausgeklammert.

[Pf98] S.L. Pfleeger: *Software Engineering - Theory and Practice*, Prentice Hall (1998), 576 Seiten

Von Idee und Aufbau her sehr schönes Buch. An dem durchgängigen Beispiel des Ariane-5-Absturzes wird potentielle Nutzen der Software-Technik demonstriert. Das Buch liefert breite Übersicht über die Software-Technik. Allerdings wird zu wenig auf objektorientierte Software-Entwicklung eingegangen.

[WO04] T. Weilkiens, B. Oestereich: *UML2 Zertifizierung - Testvorbereitung zum OMG Certified UML Professional (Fundamental)*, dpunkt.Verlag (2004), 149 Seiten

Trainingsbuch für UML-Zertifizierung; nur als Ergänzung zu anderen UML-Büchern genießbar.



1. Software-Technik - Was ist das?

Themen dieses Kapitels:

- worin unterscheidet sich Software von anderen technischen Produkten
- einige spektakuläre Softwarefehler
- Definition(en) und Geschichte des Begriffs „Software-Technik”
- Software-Qualitätsmerkmale

Merke:

Für die Entwicklung **großer Software-Produkte** mit Milliarden Zeilen Code (in Autos, Flugzeugen, Medizintechnikgeräten, ... braucht man andere Vorgehensweisen als für das Lösen kleiner Übungsaufgaben.



1.1 Bekannte Software-Katastrophen

Ein schönes Beispiel für Inkompetenz:

“Since 1989, some 120 Million Deutschmarks have been wasted trying to design and install a new computer system for the police in Hamburg, Germany. The system, designed to ease the load of paperwork on the police never worked. … What strikes me are the three conclusions of the whole affair that appeared in Hamburger Abendblatt, 16 April 1998 … ”

[ACM SIGSOFT Software Engineering Notes, vol. 23, no. 4 (1998), S. 22]

- ☹ 365 Jobs wurden gestrichen und es gab keine Pläne für Neueinstellungen. Polizeioffiziere mussten die Papierarbeit übernehmen.
- ☹ Die für das Design des Systems verantwortliche Beratungsfirma führt als einen Grund des Desasters an: ‘*police people were in charge of the development team. Software design was not one of their core competences.*’
- ☹ Der Innenminister Hartmuth Wrocklage führte die Probleme darauf zurück, dass die Komplexität des Projekts unterschätzt worden war.



Ein wirklich teurer Software-Fehler - Ariane-5:

“On June 4, 1996, on its maiden flight, the Ariane-5 was launched and performed perfectly for approximately 40 seconds. Then it began to veer off course. At the direction of the Ariane ground controller, the rocket was destroyed by remote control. … total cost of the disaster was $500 million.”

[Pfleeger: Software Engineering - Theory and Practice, S. 37]

Ursache:

- Flugbahn der Rakete wird durch “Inertial Reference System (SRI)” gemessen, dessen Software teilweise von Ariane-4 übernommen wurde
- ein Teilsystem von SRI rechnete nach dem Start weiter, obwohl seine Ergebnisse in Ariane-5 nicht mehr benötigt wurden
- andere Flugbahndaten von Ariane-5 (als bei Ariane-4) erzeugten Überlauf bei Real-Integer-Konvertierung und verursachten Fehlfunktion des SRI-Systems
- dadurch wurden wichtige Flugdaten durch ein Testmuster überschrieben
- das SRI-System und sein Backup schalteten sich aufgrund des Fehlers ab



Schlussfolgerungen aus dem Ariane-5-Beispiel:

- die Anforderungen an ein Software-System sind präzise aufzuschreiben
- es ist sicherzustellen, dass die tatsächliche Realisierung die gestellten Anforderungen (und nur diese) erfüllt
- bei Wiederverwendung von Software ist zu prüfen, ob die ursprünglichen Voraussetzungen noch gegeben sind (die dokumentiert sein sollten)
- Redundanz identischer Teilsysteme hilft gegen Hardwarefehler, nicht aber gegen Software(-Design)-fehler
- aktives Risikomanagement notwendig, das die Auswirkungen möglicher Softwarefehlfunktionen analysiert und Präventivmaßnahmen einleitet
- Software sollte nicht nur Fehlersituationen durch Konsistenzchecks erkennen, sondern robust gegen Fehlfunktionen einzelner Teilsysteme sein



1.2 Die Software-Krise von 1968 bis ?

Stand der Dinge 1968:

Programmsysteme der 60er Jahre wurden zunehmend komplexer aber es gab

- ☹ keine geeigneten (Programmier-)Sprachen
- ☹ keine geeigneten Methoden/Vorgehensweisen
- ☹ keine geeigneten Werkzeuge

Die Folgen (in der Vergangenheit?):

- ☹ Software-Kosten stiegen kontinuierlich (Hardwarekosten fallen)
- ☹ extrem viele Software-Entwicklungsprojekte scheiterten



Die Schlussfolgerung aus der Software-Krise:

- Software-Entwicklung entsprach dem Bau von Palästen ohne Architekten, Pläne, Maschinen, … .
- Erstellung von Software sollte nicht länger kreative Kunst sondern ingenieur-mäßige Wissenschaft sein mit wohldefinierten Vorgehensweisen!
- Deshalb Einführung der **Software-Technik** für die ingenieurmäßige Erstellung von Software-Systemen.

Zitat nach Dijkstra [Di72]:

„Als es noch keine Rechner gab, war auch das Programmieren noch kein Problem, als es dann ein paar leistungsschwache Rechner gab, war das Programmieren ein kleines Problem und nun, wo wir gigantische Rechner haben, ist auch das Programmieren zu einem gigantischen Problem geworden. In diesem Sinne hat die elektronische Industrie kein einziges Problem gelöst, sondern nur neue geschaffen. Sie hat das Problem geschaffen, ihre Produkte zu nutzen."



Und wo stehen wir heute:

Standish Group (http://www.standishgroup.com) veröffentlich in regelmäßigen Abständen den sogenannten „Chaos Report“ mit folgenden Ergebnissen (für 2012):

- 18% aller betrachteten IT-Projekte sind gescheitert (früher: 25%)
- 43% aller betrachteten IT-Projekte sind dabei zu scheitern (früher: 50%) (signifikante Überschreitungen von Finanzbudget und Zeitrahmen)
- 39% aller betrachteten IT-Projekte sind erfolgreich (früher: 25%)

Hauptgründe für Scheitern von Projekten:

Unklare Anforderungen und Abhängigkeiten sowie Probleme beim Änderungsmanagement!!!

„Bekannte“ Beispiele aus den letzten Jahren:

Toll-Collect, Hessische Schul-Software **LUSD**, ...



Warum ist Softwareerstellung so schwierig:

- **Kommunikationsprobleme mit dem Anwender** (wenig Anwendungswissen bei Entwicklern, in Konflikt stehende Anforderungen)
- Software ist **immateriell** und wird nicht durch physikalische Gesetze begrenzt → Modellbildung für relevanten Weltausschnitt schwierig
- Software lässt sich leicht(er) **modifizieren** als Hardware → Anforderungen ändern sich während Entwicklungszeit
- Software unterliegt keinem Verschleiss, **altert** aber trotzdem → Wartungsprobleme ("Jahr 2000"-Problem, …)
- Software muss auf verschiedenen **Plattformen** laufen → Portabilitätsprobleme
- Software existiert in vielen **Varianten** → Explosion der Variantenvielfalt
- Software greift in bestehende **Arbeitsabläufe** ein → Akzeptanzprobleme
- viele Software-Produkte sind „**einzigartig**" (keine Massenware)



Moore’s Law und Softwaretechnik:

transistors

MOORE'S LAW

Pentium® 4 Processor
Pentium® III Processor
Pentium® II Processor
Pentium® Processor
486™ DX Processor
386™ Processor
286
8086
8080
8008
4004

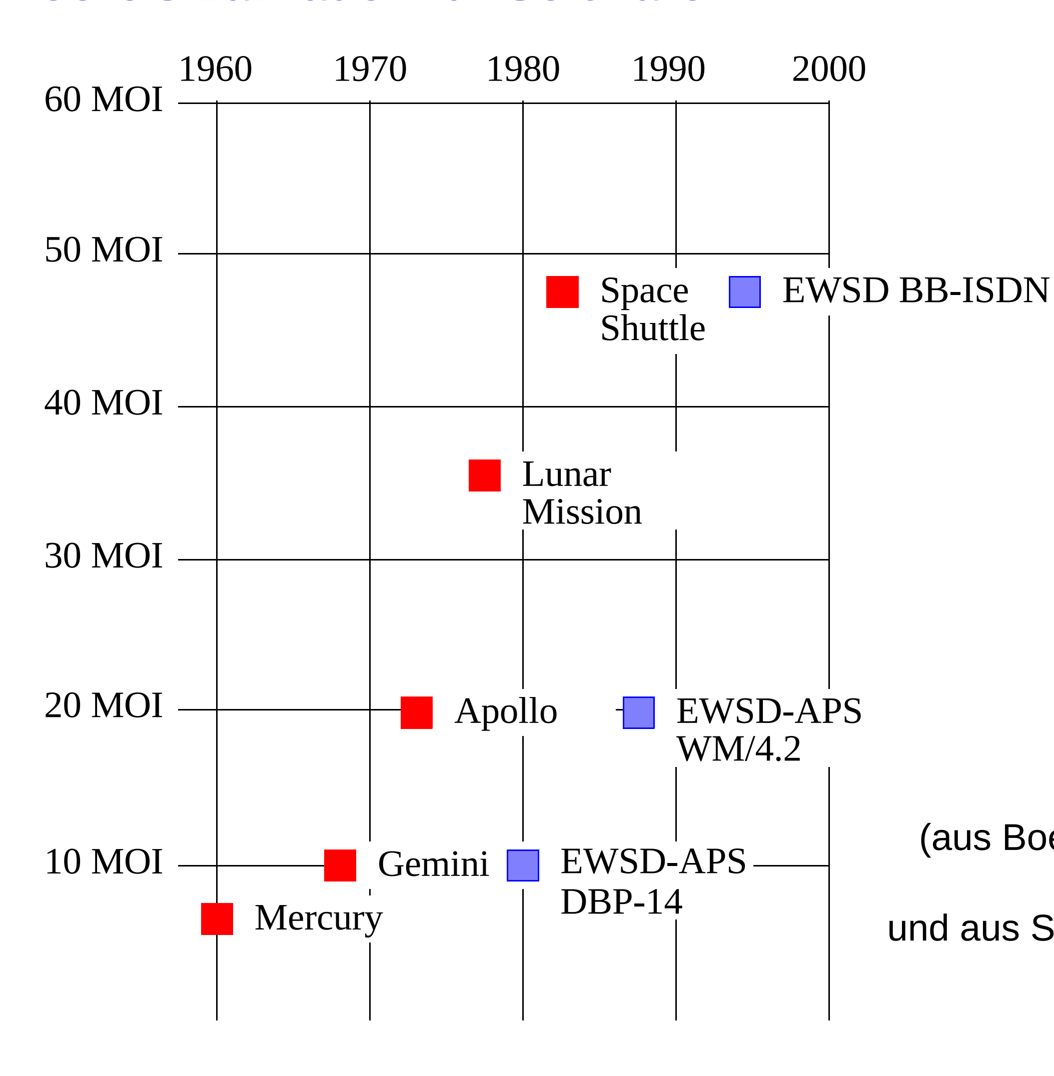
100,000,000
10,000,000
1,000,000
100,000
10,000
1000

1970
1975
1980
1985
1990
1995
2000

Quelle: http://www.intel.com/research/silicon/mooreslaw.htm



Moore's Law auch für Software?



EWSD = Elektronisches Wählsystem Digital

Software für Raumfahrt

MOI = Million Object Code Instructions

Quelle: [Ba96]
(aus Boehm: *Improving Software Productivity*, Computer (1987), 43-57
und aus Siemens: *Unterlagen zum Seminar Industrielle Software-Technik*)



Größe eingebetteter Software - 100Hz Fernseher:



Quelle: Philips (ca. 2005)

Code-Umfang:
2 Mio Zeilen Code



Größe eingebetteter Software - Mobiltelefon (Android-BS):

Code-Umfang:
12 Mio Zeilen Code

Fehlerquote:
0,78 Fehler auf 1000 Zeilen (also geschätzt ca. 9.000 Fehler)

Quelle:
Chip-Online
(2011)



Größe eingebetteter Software - Rasierapparat:



Code-Umfang:
0,1 Mio Zeilen Code

Quelle: Philips (ca. 2005)



1.3 Software-Technik - Geschichte und Definition

- Auslöser für Einführung der Software-Technik war die **Software-Krise** von 1968
- Der Begriff “**Software Engineering**” wurde von F.L. Bauer im Rahmen einer Study Group on Computer Science der NATO geprägt
- Bahnbrechend (mit Abstrichen) war die NATO-Konferenz
  “**Working Conference on Software Engineering**”
  vom 7. - 10. Oktober 1968 in Garmisch (vor 40 Jahren)



The NATO Software Engineering Conference

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html



The NATO Software Engineering Conference

http://homepages.cs.ncl.ac.uk/brian.randell/NATO/N1968/index.html



Einordung der Software-Technik:

- Das Gebiet Software-Technik wird der **praktischen Informatik** zugeordnet, hat aber auch Wurzeln in der theoretischen Informatik
- Abgrenzung zu anderen Teilgebieten der Informatik ist schwierig (wie Datenbanken, Übersetzerbau, …)
- Informatikübergreifende Aspekte spielen eine wichtige Rolle (wie Projektplanung, Organisation, …)
- Neben der Informatik gibt es eigene **Software-Technik-Studiengänge** und sogar spezielle Studiengänge wie „Automotive Software Engineering“
- Es wird auch behauptet, dass sich die

  Software-Technik zur Informatik

  wie die

  Elektrotechnik zur Physik verhält



Definition des Begriffs „Software" nach Humphry [Hu89]:

*The term software refers to a **program and all the associated information** and materials needed to support its installation, operation, repair and enhancement.*

Weniger umfassende Definition des New Oxford American Dictionary:

„software": the programs and other operating information used by a computer.

Fazit für diese Lehrveranstaltung:

Der Begriff **Software** umfasst neben dem auf einem Mikroprozessor ausführbarem Programm alle weiteren Dokumente, die für die (Weiter-)Entwicklung und Nutzung von Bedeutung sind. Dazu gehören unter anderem:

- ⇨ Beschreibung der Anforderungen an die Software
- ⇨ Modelle (Architekturen) der Software
- ⇨ alle Arten von Testfällen
- ⇨ alle Arten von (Benutzer-)Dokumentation



Definition des Begriffs „Software-Technik”:

Software Engineering = **Software-Technik** ist nach [Ka98]:

- die Entwicklung
- die Pflege und
- der Einsatz

qualitativ hochwertiger Software unter Einsatz von

- wissenschaftlichen Methoden
- wirtschaftlichen Prinzipien
- geplanten Vorgehensmodellen
- Werkzeugen
- quantifizierbaren Zielen



Ergänzende Definitionen:

- Nach [IEE83]:

  *"... the systematic approach to the development, operation, **maintenance**, and **retirement** of software."*

- Nach [BEM92]:

  *"**Software Engineering** is the science and **art** of specifying, designing, implementing, and evolving, with economy, timelineness and **elegance**, programs, documentations, and operating procedures whereby computers can be made useful to humanity."*

- Nach [Ba96]:

  *„**Software-Technik**: Zielorientierte Bereitstellung und systematische Verwendung von Prinzipien, Methoden, Konzepten, Notationen und Werkzeugen für die **arbeitsteilige**, ingenieurmäßige Entwicklung und Anwendung von **umfangreichen** Software-Systemen. Zielorientiert bedeutet die Berücksichtigung z.B. von Kosten, Zeit, Qualität."*



Definition des Begriffs „Software-Techniker“:

Software-Ingenieur = **Software-Techniker** (-Entwickler) ist nach [GJM91]:

*“A **software engineer** must of course be a good programmer, be well-versed in data structures and algorithms, and be fluent in one or more programming languages. … The software engineer must be familiar with several design approaches, be able to translate vague requirements and desires into precise specifications, and be able to converse with the user of a system in terms of application rather than ‘computeres’.”*

Daraus folgende notwendige Fähigkeiten:

- Kommunikation auf verschiedenen Abstraktionsebenen
- Kommunikation mit Personen mit unterschiedlichen Vorstellungen, Ausbildungen
- Arbeitsplanung und -koordination
- Erstellung und Verwendung von Modellen/Spezifikationen (**unser Schwerpunkt**)



Ethische Regeln und professionelles Verhalten des Software-Entwicklers:

Präambel (verkürzt):

… Software-Entwickler sollen sich verpflichten, Analyse, Spezifikation, Entwurf, Entwicklung, Test und Wartung von Software zu einem nützlichen und geachteten Beruf zu machen. In Übereinstimmung mit ihren Verpflichtungen gegenüber Gesundheit, Sicherheit und dem Wohlergehen der Öffentlichkeit sollen Software-Entwickler sich an die folgenden acht Prinzipien halten:

1. **Öffentlichkeit** — *Software-Entwickler sollen in Übereinstimmung mit dem öffentlichen Interesse handeln.*

2. **Kunde und Arbeitgeber** — *Software-Entwickler sollen auf eine Weise handeln, die im Interesse ihrer Kunden und ihres Arbeitgebers ist und sich mit dem öffentlichen Interesse deckt.*

3. **Produkt** — *Software-Entwickler sollen sicherstellen, dass ihre Produkte und damit zusammenhängende Modifikationen den höchstmöglichen professionellen Standards entsprechen.*



Ethische Regeln - Fortsetzung:

4. **Beurteilung** — *Software-Entwickler sollen bei der Beurteilung eines Sachverhalts Integrität und Unabhängigkeit bewahren.*

5. **Management** — *Für das Software Engineering verantwortliche Manager und Projektleiter sollen sich bei ihrer Tätigkeit ethischen Grundsätzen verpflichtet fühlen und in diesem Sinne handeln.*

*6.***Beruf** — *Software-Entwickler sollen die Integrität und den Ruf des Berufs in Übereinstimmung mit dem öffentlichen Interesse fördern.*

7. **Kollegen** — *Software-Entwickler sollen sich ihren Kollegen gegenüber fair und hilfsbereit verhalten.*

8. **Selbst** — *Software-Entwickler sollen sich einem lebenslangen Lernprozess in Bezug auf ihren Beruf unterwerfen und anderen eine ethische Ausübung ihres Berufes vorleben.*

aus „ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices“ - sinngemäße Übersetzung aus [So07]



1.4 Software-Qualitätsmerkmale

Ziel der Software-Technik:

Effiziente Entwicklung **messbar** qualitativ hochwertiger Software, die

- ⇨ korrekt bzw. zuverlässig arbeitet
- ⇨ robust auf Fehleingaben, Hardwareausfall, … reagiert
- ⇨ effizient (ökonomisch) mit Hardwareressourcen umgeht
- ⇨ benutzerfreundliche Oberfläche besitzt
- ⇨ wartbar und wiederverwendbar ist

Wir unterscheiden:

- ⇨ **externe Qualitätsfaktoren**: sichtbar für Benutzer des Systems
- ⇨ **interne Qualitätsfaktoren**: sichtbar für Entwickler des Systems

Achtung: es gibt auch die **Qualität** des Software-Entwicklungs-**Prozesses**.



Beispiel für fehlende Software-Qualitäten:

```
program SORT;
var a, b: file of integer;
      Feld: array [ 1 .. 10 ] of integer;
      i, j, k, l : integer
begin
      open ( a, '/usr/schuerr/Zahlen/Datei'); i := 1;
      while not eof ( a ) do
      begin
            read ( a, Feld [ i ] ); i := i + 1
      end;
      l := i - 1;
      open ( b, '/usr/schuerr/Zahlen/Datei');
      for i := 2 to l do
            begin
                  for j := l downto i do
                        if Feld [ j - 1 ] > Feld [ j ] then
                        begin
                              k := Feld [j - 1]; Feld [j - 1] := Feld[ j ]; Feld [ j ] := k
                        end;
                        write ( b , Feld [ i - 1 ] )
            end
end SORT;
```



Bewertung des Sortierprogramms:

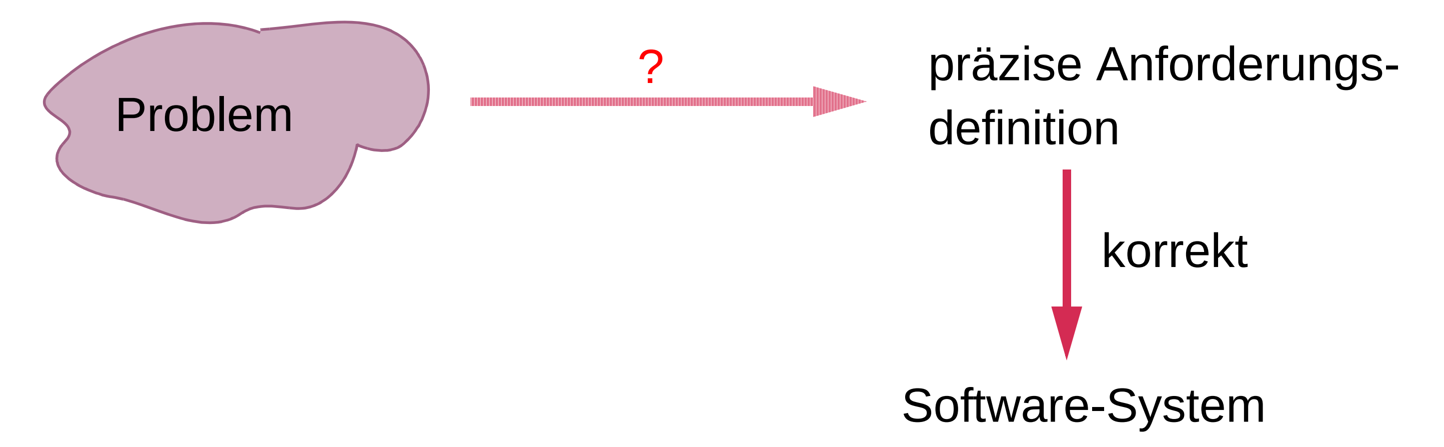
- ☹ nicht leicht verständlich, da:
  - ⇨ Kommentare fehlen, Variablenbezeichner nichtssagend, …
- ☹ nicht korrekt (robust), da:
  - ⇨ Programm für Dateien mit mehr als 10 Elementen abstürzt
- ☹ nicht effizient, da:
  - ⇨ realisierter Sortieralgorithmus quadratischen Aufwand hat
- ☹ nicht portierbar, da:
  - ⇨ Dateinamenskonventionen von Unix im Quelltext auftauchen
  - ⇨ dieselbe Datei mehrfach geöffnet wird (geht nicht immer)
- ☹ nicht wiederverwendbar, da:
  - ⇨ Dateinamen “fest verdrahtet” sind
  - ⇨ ebenso maximale Dateilänge und zu sortierende Elemente



Korrektheit von Software (Correctness):

Ein Software-System ist (funktional) korrekt, wenn es sich genauso verhält, wie es in der **Anforderungsdefinition** festgelegt wurde.

Korrektheit ist also relativ:



Großes Problem:

☹ Anforderungsdefinitionen sind häufig informal und damit eigentlich nie ganz widerspruchsfrei, vollständig, präzise formuliert, …



Verfügbarkeit von Software (Availability):

Verfügbarkeit ist ein Maß dafür, wie häufig ein Software-System (nicht) die gewünschte Funktionalität (in vollem Umfang) zur Verfügung stellen kann.

Metriken für Bewertung der Verfügbarkeit von Software:

1. **„rate of failure occurrence" (ROFOC)**: Häufigkeit von nicht erwartetem Verhalten, z.B. 2 Fehler pro 100 operationellen Zeiteinheiten

2. **„mean time to failure" (MTTF)**: mittlerer Zeitabstand zwischen zwei Fehlern (also von unerwartetem bzw. unerwünschtem Verhalten)

   Problem: Schwere der Fehler und (Zeit-)Aufwand für Fehlerbehebung werden bei obigen Definitionen erst mal nicht berücksichtigt

3. **„availability" (AVAIL)**: mittlere Verfügbarkeit der Software, z.B. 998 von 1000 Zeiteinheiten war das System benutzbar

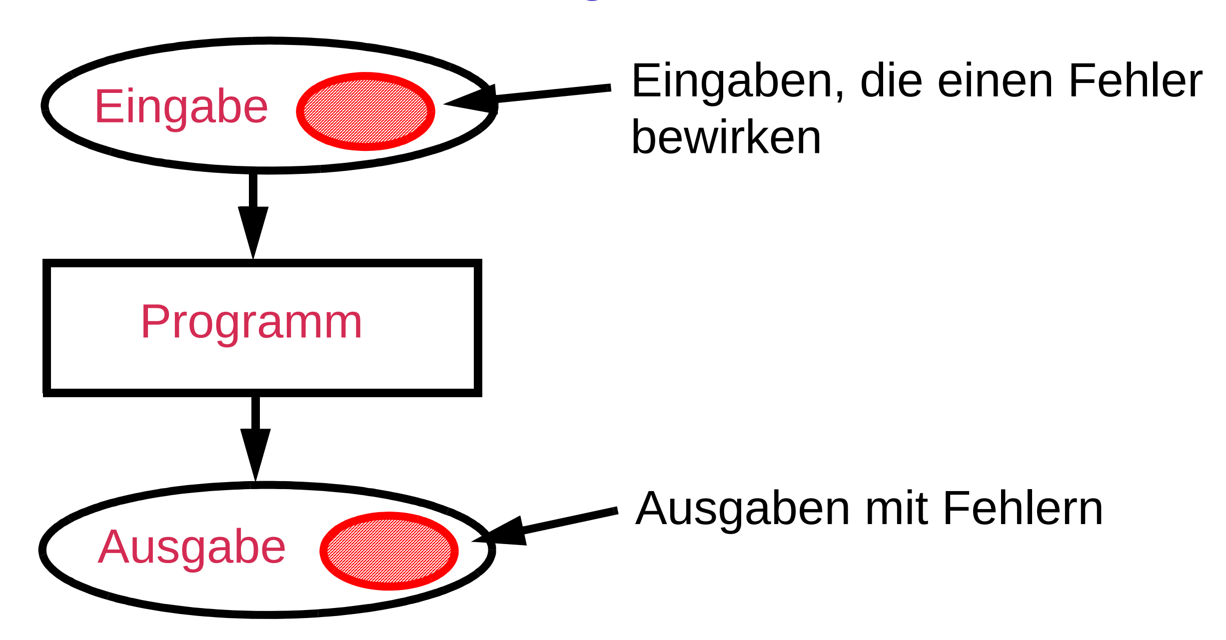
   Problem: Berücksichtigt (Zeit-)Aufwand für Fehlerbehebungen oder andere Wartungsaktivitäten, aber nicht Schwere der Fehler.



Zuverlässigkeit von Software (Reliability):

Zuverlässige Software **funktioniert meistens**, Zuverlässigkeit ist also ein Maß für die Wahrscheinlichkeit, dass ein Software-System sich in genau so verhält, wie es von ihm erwartet wird.

Korrekte Software = 100% zuverlässige Software:





Vertrauenswürdige Software (Safety):

Vertrauenswürdige Software verursacht (auch) im Fehlerfall **keine Katastrophen**.

- ⇨ inkorrekte Software kann vertrauenswürdig sein
  (z.B. wenn die Software per se keinen wirklichen Schaden anrichten kann)
- ⇨ korrekte Software muss nicht vertrauenswürdig sein
  (Fehler in Anforderungsdefinition)

Beispiele (nicht) vertrauenswürdiger Software:

- ❑ Joystick-Steuerung in Spielconsole: von Natur aus vertrauenswürdig
- ❑ Joystick-Steuerung im Los Alamos National Laboratory:

  Am 26. Februar 1998 führte die Fehlfunktion einer Joystick-Steuerung dazu, dass sich zwei Uranstücke nicht langsam, sondern mit maximaler Geschwindigkeit aufeinander zubewegten. Der Operator drückte rechtzeitig einen Notausknopf (angeblich war die Summe der Massen unterkritisch).

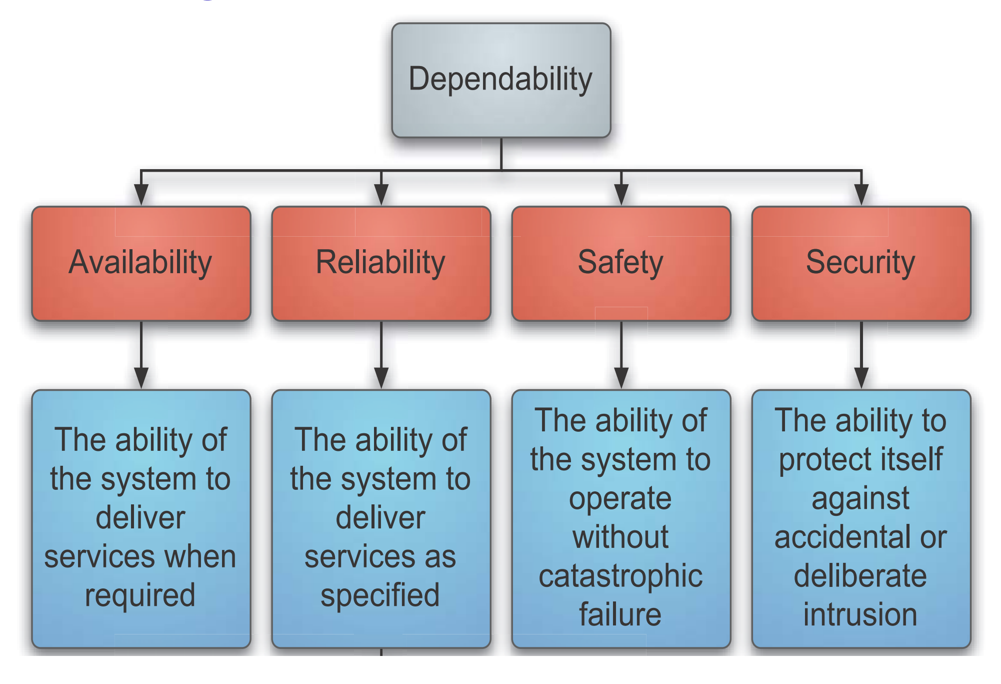
  [ACM SIGSOFT Sofware Engineering Notes, vol. 23, no. 4 (1998), S. 21]



Sichere Software (Security):

Software sollte gegen „böswillige“ Fehlbenutzungen und Angriffe abgesichert sein. Dabei besteht ein fließender Übergang zwischen der Absicherung eines Software-Systems gegen unabsichtliche Ausfälle von Komponenten oder Fehlbedienungen (siehe Robustheit) und absichtliche Angriffe.

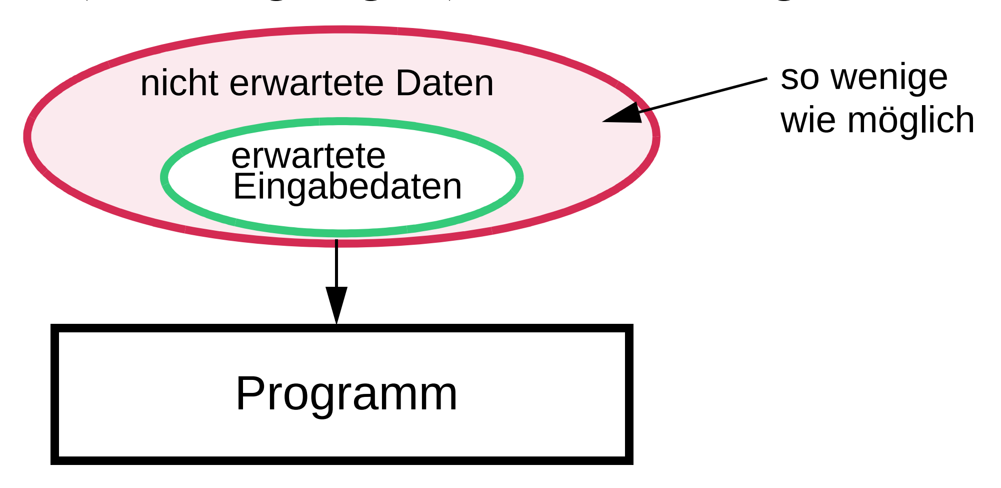
Dependability als Oberbegriff nach Sommerville [So07]:





Robuste Software (Robustness):

Ein Software-System ist robust, wenn es auch unter **unvorhergesehenen Umständen** funktioniert (vernünftig reagiert), z.B. auf zufällige oder beabsichtige Angriffe.



Anmerkung:

Auch der Software-Entwicklungsprozess sollte robust sein, z.B. gegen

- ⇨ Ausfall von Mitarbeitern
- ⇨ unvermeidlicher Hardware-/Betriebssystemwechsel



Interne Qualitätsfaktoren von Software:

- Effizienz (Efficiency, Performance)
- Erweiterbarkeit (Extendibility)
- Kompatibilität mit anderen Komponenten (Compatibility)
- Portierbarkeit auf andere Plattformen (Portability)
- Wartbarkeit (maintenability)
- Wiederverwendbarkeit (Reusability)
- ...



Effiziente Software (Efficiency):

Effiziente Software nutzt Hardware-Ressourcen **(hinreichend) ökonomisch** (sodass auf der verfügbaren Hardware die Software benutzbar ist).

Benötigte Ressourcen:



Wie ermittelt man Effizienz:

- theoretische Komplexitätsanalyse („O-von“-Rechnungen)
- Simulationsmodelle
- Messungen am realen System (profiling, …)



Effizienz des Software-Erstellungsprozesses:

Effiziente Software-Herstellung = **produktive** Software-Herstellung, also mit möglichst wenig Mitarbeitern in möglichst kurzer Zeit möglichst **gute** Software herstellen.

Problem:



Lösung:

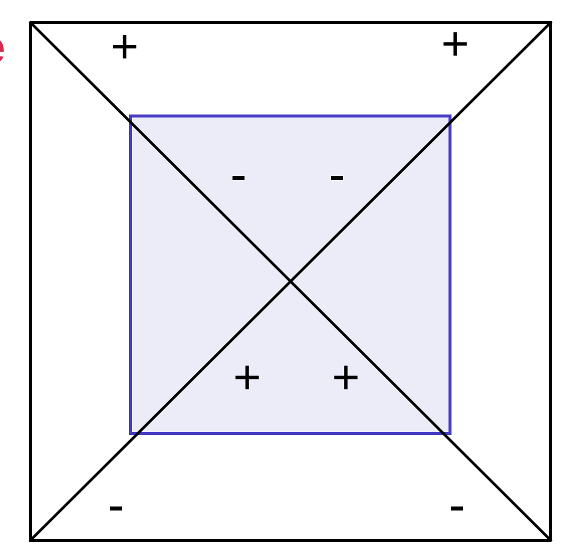
- **Wiederverwendung** von Software-Komponenten, Vorgehensweisen, …
- Einsatz von sehr hohen Programmiersprachen, Codegeneratoren
- …

Wie misst man Produktivität:

- in keinem Fall durch LOC = “Lines of Code”



Das Teufelsquadrat nach Sneed [Sn87]:

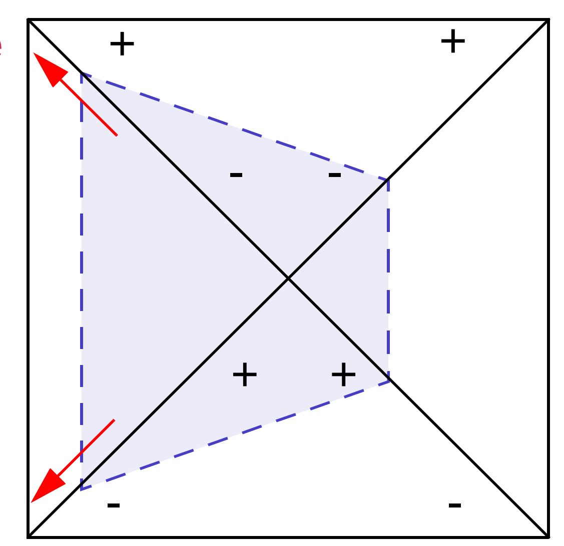


Erläuterungen zum Teufelsquadrat:

- für Software-Entwicklungsprozess werden Qualität und Quantität der Software sowie Entwicklungsdauer und -kosten gemessen
- die Messwerte werden auf den vier Skalen (von den Ecken zur Mitte verlaufend) eingetragen und die Punkte miteinander verbunden



Das Teufelsquadrat nach Sneed [Sn87]:



Erläuterungen zum Teufelsquadrat:

- Fläche des Quadrats = **Produktivität** ist invariant (Veränderung ggf. durch Mitarbeiterschulung, besseres Vorgehensmodell, …)
- Erhöhung der Qualität und Reduktion der Entwicklungsdauer geht nur mit Reduktion des Produktumfanges und/oder Erhöhung der Entwicklungskosten



Wartbarkeit von Software (Maintenability):

Software-Wartung = **Änderungen** an der Software aufgrund von Fehlern oder geänderten Anforderungen. Gut wartbare Software ist leicht korrigierbar, modifizierbar und erweiterbar.

Wartbarkeit wird unterstützt durch:

- gute Systemstruktur (Modularisierung der Software)
- gute Dokumentation
- kontrollierte Änderungsprozeduren

Achtung:

Jede Wartung reduziert die Wartbarkeit, solange bis die Software nicht mehr änderbar ist (Gesetz der zunehmenden Entropie).



Portierbare Software (Portability):

Ein System ist portierbar, falls es in **verschiedenen Umgebungen** (ohne großen Änderungsaufwand) läuft.

Umgebung = Hardware und/oder Basissoftware (Betriebssystem, Fenstersystem, …)

Portierbarkeit erreicht man durch:

- Trennung (kleiner) umgebungsabhängiger Teile vom Rest des Systems
- Verwendung standardisierter (Programmier-)Sprachen
- Verwendung standardisierter (weitverbreiteter) Betriebssysteme
- Verwendung standardisierter Fenstersysteme



1.5 Wissensgebiete der Software-Technik

Der IEEE Computer Society „Guide to the Software Engineering Body of Knowledge" (SWEBOX, http://www.swebox.org) zählt folgende Wissensgebiete auf:

1. **Software Requirements**:

   es wird festgelegt, „**was**" ein Software-System leisten soll (und warum)

2. **Software Design**:

   das „**Wie**" steht nun im Vordergrund, der Bauplan (Architektur)

3. **Software Construction**:

   gemäß Bauplan wird das Software-System realisiert

4. **Software Testing**:

   Fehler werden **systematisch** gesucht und eliminiert



5. **Software Maintenance**:

   die Pflege und Weiterentwicklung der Software nach Auslieferung

6. **Software Configuration Management**:

   die Verwaltung von Software-Versionen und -Konfigurationen

7. **Software Engineering Management**:

   (Projekt-)Management von Personen, Organisationen, Zeitplänen, …

8. **Software Engineering Process**:

   Definition und Verbesserung von Software-Entwicklungsprozessen

9. **Software Engineering Tools and Methods**:

   Werkzeuge und Methoden für die Software-Entwicklung

10. **Software Quality**:

    Messen und Verbessern der Software-Qualität



Methoden in der Softwaretechnik:

- Richtlinien zum Umgang mit einzelnen Sprachen
- Richtlinien zum Umgang mit einzelnen Werkzeugen

Werkzeuge (Computer Aided Software Engineering = CASE Tools):

- **Editoren**: Texteditor, syntaxgestützter Editor, Diagrammeditor
- **Ausführungswerkzeuge**: Übersetzer, Interpreter
- **Testwerkzeuge**: Debugger, Testdatengenerator, Testrahmengenerator
- **Programmanalysewerkzeuge**: statische Analysen, dynamische Analysen
- …

Vorgehensmodell:

Vereinfachte Beschreibung eines Software-Entwicklungsprozesses mit Regeln zum koordinierten Einsatz von Werkzeugen, Methoden und Sprachen



1.6 Weitere Literatur

[BD00] B. Bruegge, A.H. Dutoit: *Object-Oriented Software Engineering: Conquering Complex and Changing Systems*, Prentice Hall (2000)

Mit seiner kurzen Einführung in UML und einer breiten Darstellung von Kommunikationsformen, Managementstrategien, Analyse- und Designheuristiken eine brauchbare Ergänzung zu reinen UML-Darstellungen. Im Gegensatz zu Standard-Software-Technik-Büchern werden Themen wie Testen, etc. aus OO-Perspektive behandelt.

[BEM92] A.W. Brown, A.N. Earl, J.A. McDermid: *Software Engineering Environments: Automated Support for Software Engineering*, McGraw-Hill (1992)

Etwas trockenes und nicht mehr ganz aktuelles Buch zum Thema Software-Werkzeuge. Beschreibt sehr schön Anforderungen an solche Umgebungen, liefert aber keine Informationen zu aktuellen CASE-Tools.

[Di72] E.W. Dijkstra: *The Humble Programmer*, Communications of the ACM, Vol. 15, No. 10 (1972)

Aufsatz von historischem Interesse.

[GJM91] C. Ghezzi, M. Jazayeri, D. Mandrioli: *Fundamentals of Software Engineering*, Prentice Hall (1991)

Klassisches Software-Technikbuch mit stärker lehrbuchartigem Charakter (im Vergleich zu [Ba96]). Naturgemäß leider nicht mehr auf der Höhe der Zeit, trotzdem aber lesenswert.

[Hu89] W.S. Humphry: *The Software Engineering Process: Definition and Scope*; ACM SIGSOFT Software Engineering Notes, Vol. 14, No. 4 (1989)

Humphry ist einer der „Großen“ im Bereich der Verbesserung von Softwareentwicklungsprozessen. Seine Bücher und Aufsätze sind durchweg empfehlenswert für die Fortbildung!



[IEE83] IEEE: *Standard Glossar of Software Engineering Terminology - IEEE Standard 729*, IEEE Computer Society Press (1983)

Der Titel sagt bereits alles.

[Ka98] B. Kahlbrandt: *Software-Engineering: Objektorientierte Software-Entwicklung mit der Unified Modeling Language*, Springer Verlag (1998)

Mit das Buch, das vom Inhalt her dieser Vorlesung am nächsten steht. Der Schwerpunkt liegt auf dem Einsatz von UML; allgemeinere Themen wie Kostenschätzung und Testen werden dafür ausgeklammert.

[Sn87] H.M. Sneed: *Software-Management*, Müller GmbH (1987)

Quelle zum Teufelsquadrat (Abhängigkeit von Kosten, Qualität, Quantität und Entwicklungszeit).



2. Vorgehensmodelle der Software-Entwicklung

Themen dieses Kapitels:

- Lebensabschnitte der Software-Entwicklung und -Alterung
- das „traditionelle“ Vorgehensmodell zur Software-Entwicklung
- wichtige Begriffe wie „Lastenheft“, „Anforderungsdefinition“, …

Merke:

Voraussetzung für den sinnvollen Einsatz von Notationen und Werkzeugen zur Software-Entwicklung ist ein **Vorgehensmodell**, das den Gesamtprozess der Software-Erstellung und -pflege in einzelne Schritte aufteilt und die Verantwortlichkeiten der beteiligten Personen (**Rollen**) klar regelt.



2.1 Der Code-and-Fix-Zyklus

Typische Vorgehensweise für kleine 1-Personen-Projekte und Übungsaufgaben im (Grund-)Studium:

1. Code schreiben und übersetzen
2. Code testen (debuggen)
3. Code „verbessern“ (Fehlerbeseitigung, Erweiterung, Effizienzsteigerung, …)
4. GOTO 2

Probleme mit dieser Vorgehensweise:

- ☹ Wartbarkeit und Zuverlässigkeit nehmen kontinuierlich ab
- ☹ wenn der Programmierer kündigt ist alles vorbei
- ☹ wenn Entwickler und Anwender nicht identisch sind, gibt es oft Meinungsverschiedenheiten über erwarteten/realisierten Funktionsumfang



2.2 Das „klassische" Wasserfallmodell

Einteilung in Phasen

Phasen erzeugen Dokumente

- Machbarkeits-studie — siehe Kapitel 3 [und Kapitel 10]
- Anforderungs-analyse — siehe Kapitel 5
- System-entwurf, Codieren und Modultest — siehe Kapitel 6 bis Kapitel 8
- Integration & Systemtest — [siehe Kapitel 9] [siehe Kapitel 10]
- Auslieferung & Installation
- Wartung

Integration bis Wartung: siehe auch andere LVs

Entwicklungszeit



Machbarkeitsstudie (feasibility study, Projektdefinition):

Die Machbarkeitsstudie schätzt **Kosten und Ertrag der** geplanten **Software-Entwicklung** ab. Dazu grobe Analyse des Problems mit Lösungsvorschlägen.

- **Aufgaben** (siehe auch Abschnitt 3.2):
  - ⇨ Problem informell und abstrahiert beschreiben
  - ⇨ verschiedene Lösungsansätze erarbeiten
  - ⇨ Kostenschätzung durchführen
  - ⇨ Angebotserstellung
- **Ergebnisse**:
  - ⇨ Lastenheft = (sehr) grobes Pflichtenheft
  - ⇨ Projektkalkulation
  - ⇨ Projektplan
  - ⇨ Angebot an Auftraggeber



Anforderungsanalyse (requirements engineering):

In der Anforderungsanalyse wird exakt festgelegt, **was die Software leisten soll**, aber nicht wie diese Leistungsmerkmale erreicht werden.

- **Aufgaben** (siehe auch Kapitel 5):
  - ⇨ genaue Festlegung der Systemeigenschaften wie Funktionalität, Leistung, Benutzungsschnittstelle, Portierbarkeit, … im Pflichtenheft
  - ⇨ Bestimmen von Testfällen
  - ⇨ Festlegung erforderlicher Dokumentationsdokumente
- **Ergebnisse**:
  - ⇨ Pflichtenheft = Anforderungsanalysedokument
  - ⇨ Akzeptanztestplan
  - ⇨ Benutzungshandbuch (1-te Version)



Systementwurf (system design/programming-in-the-large):

Im Systementwurf wird exakt festgelegt, wie die Funktionen der Software zu realisieren sind. Es wird der **Bauplan der Software**, die Software-Architektur, entwickelt

- **Aufgaben** (siehe auch Kapitel 6, Kapitel 7 und Kapitel 8):
  - Programmieren-im-Großen = Entwicklung eines Bauplans
  - Grobentwurf, der System in Teilsysteme/Module/Pakete zerlegt
  - Auswahl bereits existierender Software-Bibliotheken, Rahmenwerke, …
  - Feinentwurf, der Modulschnittstellen und Algorithmen vorgibt
  - konzeptuelles Design von Datenbankstrukturen
- **Ergebnisse**:
  - Entwurfsdokument mit Software-Bauplan
  - detailierte(re) Testpläne



Codieren und Modultest (Implementierung, programming-in-the-small):

Die eigentliche **Implementierungs- und Testphase**, in der einzelne Module (in einer bestimmten Reihenfolge) realisiert und validiert werden.

- **Aufgaben** (siehe auch Kapitel 8 und Kapitel 9):
  - ⇨ Programmieren-im-Kleinen = Implementierung einzelner Module
  - ⇨ Einhaltung von Programmierrichtlinien
  - ⇨ Code-Inspektionen kritischer Modulteile (Walkthroughs)
  - ⇨ Test der erstellten Module
- **Ergebnisse**:
  - ⇨ Menge realisierter Module
  - ⇨ Implementierungsberichte (Abweichungen vom Entwurf, Zeitplan, …)
  - ⇨ technische Dokumentation einzelner Module
  - ⇨ Testprotokolle



Integration und Systemtest (Systemintegration):

Die einzelnen Module werden schrittweise zum **Gesamtsystem zusammengebaut**. Diese Phase kann mit der vorigen Phase verschmolzen werden, falls der Test isolierter Module nicht praktikabel ist.

- **Aufgaben** (siehe Kapitel 9):
  - Systemintegration = Zusammenbau der Module
  - Gesamtsystemtest in Entwicklungsorganisation durch Kunden (α-Test)
  - Fertigstellung der Dokumentation
- **Ergebnisse**:
  - fertiges System
  - Benutzerhandbuch
  - technische Dokumentation
  - Testprotokolle



Auslieferung und Installation (Migration und Einführung):

Die Auslieferung (Installation) und **Inbetriebnahme** der Software **beim Kunden** findet häufig in zwei Phasen statt.

- **Aufgaben** (siehe LV „Software Engineering - Wartung und Qualitätssicherung"):
  - ⇨ Auslieferung an ausgewählte (Pilot-)Benutzer (β-Test)
  - ⇨ Auslieferung an alle Benutzer (Migration auf neues System)
  - ⇨ Schulung bzw. Einführung der Benutzer
- **Ergebnisse**:
  - ⇨ fertiges System
  - ⇨ Akzeptanztestdokument



Wartung (Maintenance):

Nach der ersten Auslieferung der Software an die Kunden beginnt im Zuge des Betriebs der Software das Elend der Software-Wartung, das ca. 60% der gesamten Software-Kosten ausmacht.

- **Aufgaben** (siehe LV „Software Engineering - Wartung und Qualitätssicherung“):
  - ca. 20% Fehler beheben (corrective maintenance)
  - ca. 20% Anpassungen durchführen (adaptive maintenance)
  - ca. 50% Verbesserungen vornehmen (perfective maintenance)
- **Ergebnisse**:
  - Software-Problemberichte (bug reports)
  - Software-Änderungs- und Renovierungsvorschläge
  - neue Software-Versionen

Die Wartungsphase endet mit der **Stilllegung** der Software (und Ersatz durch vollständig neue Software)



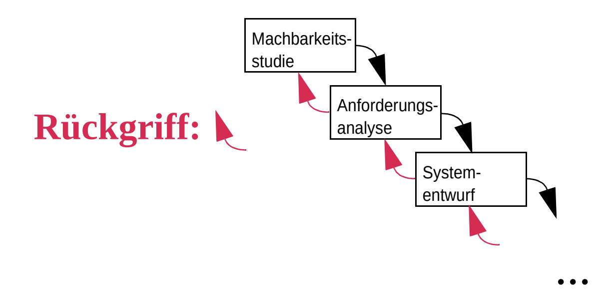
Probleme mit dem Wasserfallmodell:

- ☹ zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
- ☹ ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erst sehr spät entsteht (Risikomaximierung)
- ☹ es gibt Fälle, in denen zu Projektbeginn kein vollständiges Pflichtenheft erstellt werden kann (weil Anforderungen nicht klar)
- ☹ Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, … Änderungen) nicht eingeplant
- ☹ Abnehmer der Software können erst das fertige Produkt bewerten
- ☹ strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
- ☹ Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase
- ☹ grundsätzlichere Renovierungsmaßnahmen und endgültige Stilllegung (Ablösung) der Software werden meist nicht mitgeplant



2.3 Andere Vorgehensmodelle

Die naheliegendste Idee zur Verbesserung des Wasserfallmodells ergibt sich durch die Einführung von **Zyklen** bzw. **Rückgriffen**. Sie erlauben Wiederaufnehmen früherer Phasen, wenn in späteren Phasen Probleme auftreten.



Andere Beispiele:

- das evolutionäre Modell (iteriertes Wasserfallmodell)
- „Rapid Prototyping“ (für Vorstudien)
- das V-Modell (XT) der Bundesbehörden (Bundeswehr)
- … (siehe Kapitel 10 der Vorlesung)



Rollenbasierte Software-Entwicklung und Arbeitsbereiche:

- Projekte funktionieren, wenn die richtigen Personen unter Verwendung geeigneter Sprachen, Methoden und Werkzeuge **vernünftig** miteinander arbeiten
- Zuweisung von Rollen an Personen klärt Verantwortlichkeiten und Kompetenzen
- Rollen und Arbeitsbereiche legen (noch) nicht fest, in welcher Reihenfolge Aufgaben erledigt werden

Begriffsfestlegungen:

- eine **Rolle** beschreibt eine Menge von eng zusammengehörigen Aufgaben und Verantwortlichkeiten (oft auch notwendige Qualifikationen)
- ein **Arbeitsbereich** umfasst eine Menge eng zusammengehöriger Aufgaben und Verantwortlichkeiten
- zu einem Arbeitsbereich gehört eine Menge voneinander abhängiger **Artefakte**, die als Eingabe benötigt oder als Ausgabe produziert werden
- oft besteht eine 1-zu-1-Beziehung zwischen Rollen und Arbeitsbereichen



Unvollständiges Beispiel für Rollenverteilung:



2.4 Weitere Literatur

[Be99] K. Beck: *Extreme Programming Explained*, Addison Wesley (1999)

Die Einführung in das Thema XP geschrieben vom Erfinder/Papst selbst. Selbst habe ich das Buch nie in Händen gehalten und werde es auch in Zukunft nicht lesen.

[BEM92] A.W. Brown, A.N. Earl, J.A. McDermid: *Software Engineering Environments: Automated Support for Software Engineering*, McGraw-Hill (1992)

Etwas trockenes und nicht mehr ganz aktuelles Buch zum Thema Software-Entwicklungsumgebungen. Beschreibt Anforderungen an Umgebungen, liefert aber keine Informationen zu aktuellen CASE-Tools.

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*, Addison Wesley (1999)

Präsentation des auf die UML zugeschnittenen Vorgehensmodells der Software-Entwicklung, eine Variante des hier vorgestellten Rational Unified (Objectory) Prozesses.

[OHJ99] B. Oestereich (Hrsg.), P. Hruschka, N. Josuttis, H. Kocher, H. Krasemann, M. Reinhold: *Erfolgreich mit Objektorientierung: Vorgehensmodelle und Managementpraktiken für die objektorientierte Software-Entwicklung*, Oldenbourg Verlag (1999)

Ein von Praktikern geschriebenes Buch mit einer Fülle von Tipps und Tricks.



3. Requirements Engineering und Machbarkeitsstudie

Themen dieses Kapitels:

- Vorbereitung und Planung eines Software-Entwicklungsprojektes
- Methoden der Anforderungsbestimmung (Requirements Engineering)
- Machbarkeitsstudie für Software-Entwicklungsprojekt
- [Schätzmethoden für Produktumfang und Entwicklungskosten]

Achtung:

Einige Themen dieses Kapitels - insbesondere die Projektplanung und Kostenschätzung - werden hier nur angerissen. Eine Vertiefung der Themen erfolgt im Kapitel 10 der Vorlesung.



Beispiel „Kundenbeschreibung eines Softwareprodukts“:

Motor Vehicle Reservation System (MVRS)

A rental office lends motor vehicles of different types. The assortment comprises cars, vans, and trucks. Vans are small trucks, which may be used with the same driving license as cars. Some client may reserve motor vehicles of a certain category for a certain period. He or she has to sign a reservation contract. The rental office guarantees that a motor vehicle of the desired category will be available for the requested period. The client may cancel the reservation at any time. When the client fetches the motor vehicle he or she has to sign a rental contract and optionally an associated insurance contract. Within the reserved period, at latest at its end, the client returns the motor vehicle and pays the bill.



3.1 Die Durchführbarkeitsprüfung für ein Software-Projekt

Wie bei jeder anderen Produktentwicklung auch muss vor Beginn eines Software-Entwicklungsprojektes in einer **Machbarkeitsstudie** (Durchführbarkeitsstudie) geprüft werden, ob

- ⇨ die **ökonomische** Durchführbarkeit
- ⇨ die **fachliche** Durchführbarkeit
- ⇨ die **personelle** Durchführbarkeit

für verschiedene Realisierungsalternativen gegeben ist (Risikoabschätzung!).

Dabei ist zu unterscheiden zwischen Software-Produktion für

- ⇨ **konkreten Auftraggeber**: generell besteht Bedarf für Software-Produkt, zu klären ist „nur" was der Auftraggeber genau haben will.
- ⇨ **anonymen Markt**: es sind Trendstudien, Marktanalysen, etc. durchzuführen, um Bedarf und Rentabilität zu klären.



Ökonomische Durchführbarkeit:

Das „Zauberwort“:

ROI = Return On Investment

Die Software-Entwicklung muss sich lohnen, sprich die Einsparungen/Gewinne durch den Einsatz der neuen Software müssen höher als die (geschätzten) Kosten der Software-Entwicklung sein.

Trifft nicht zu auf:

- Änderungen auf Grund neuer gesetzlicher Bestimmungen (z.B. geänderter Mehrwertsteuersatz, Datenschutzbestimmungen)
- Ablösung veralteter Technologien (z.B. Hardware-Plattformen, Programmiersprachen)
- …



Bestandteile einer Machbarkeitsstudie:

- **Lastenheft**: führt alle fachlichen Anforderungen grob auf, die Software aus Sicht des Auftraggebers erfüllen soll (es wird bei klassischer Vorgehensweise nicht festgelegt, wie sich die Anforderungen realisieren lassen).
- **Projektkalkulation**: der Umfang des gewünschten Software-Produktes wird geschätzt und es werden daraus die Entwicklungskosten abgeleitet.
- **Projektplan**: Zeitplan für die Durchführung des Projektes mit Unterteilung in Phasen und Festlegung von Phasenergebnissen und Phasenverantwortlichen.

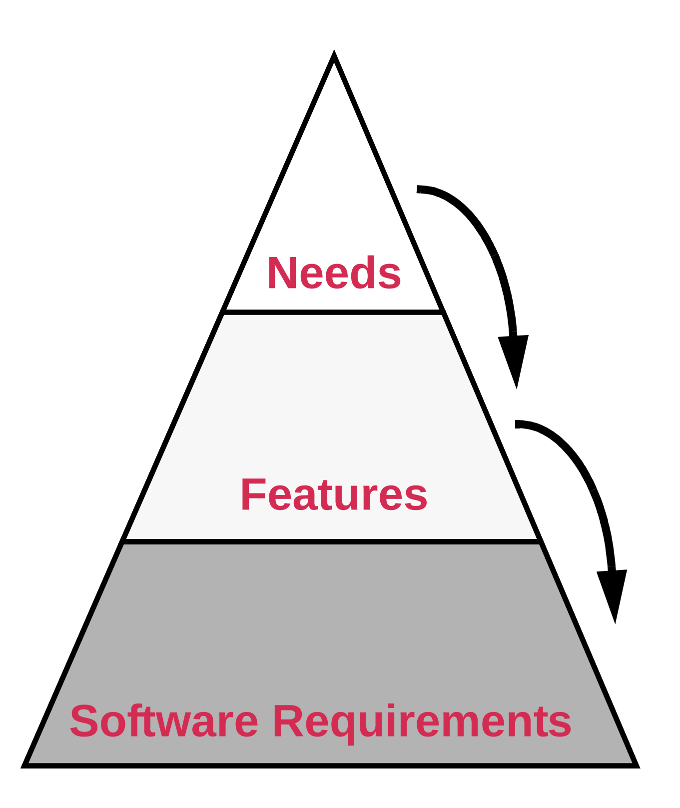
Große Probleme:

- wie ist Umfang/Größe eines Software-Produktes definiert
- wie läßt sich Produktumfang ohne Betrachtung der Realisierung schätzen
- in welchem Verhältnis stehen Produktumfang, Entwicklungszeit und -kosten



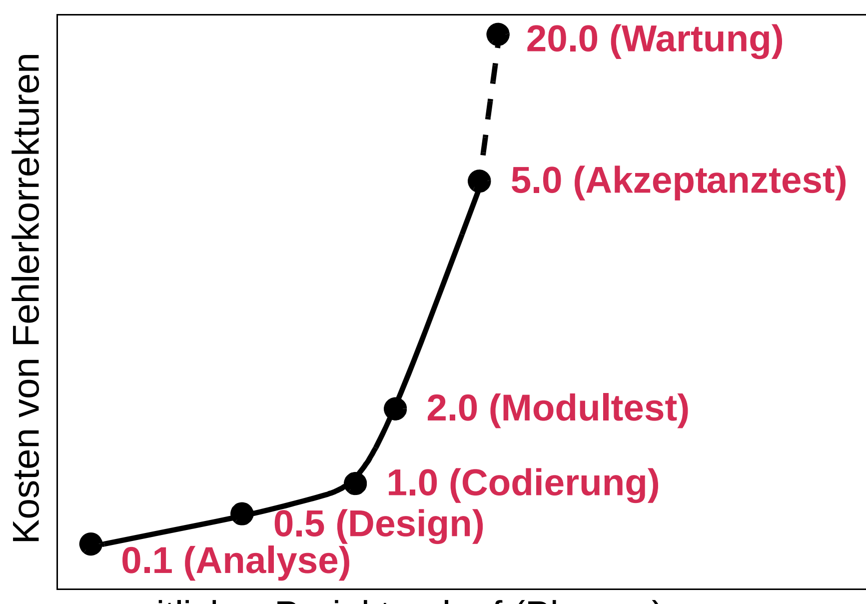
Ermittlung von Anforderungen aus [WW00]:

- **Needs** = Bestandsaufnahme existierender Geschäftsprozesse, Problembeschreibungen: **warum** soll Software entwickelt werden
- **Features** = **Lastenheft** mit grober Beschreibung von Hauptfunktionen neuer Software: **was** für Funktionen sollen die gefundenen Probleme lösen (siehe Abschnitt 3.3)
- **Software Requirements** = **Pflichtenheft** mit detaillierter Beschreibung der Aufgaben der neu zu entwickelnden Software: immer noch steht „**was** für Funktionalität benötigt wird“ und nicht „**wie** die gewünschte Funktionalität zu realisieren ist“ im Vordergrund (siehe Kapitel 5)





Anzahl und Auswirkung von Anforderungsfehlern [WW00]:



In Phase X beseitigte Fehler können aus Phase X stammen oder Folgefehler von Fehlern in einer früheren Phase sein.

Fazit:

Fast 1/4 aller Anforderungsfehler bleibt bis zur Auslieferung unentdeckt. Die Beseitigung dieser Fehler bei der Wartung (statt bei der Analyse) ist 200 Mal so teuer … .

Fehlerursprung	relative Fehlerzahl	Eliminationsrate	ausgelieferte Fehler
Analyse	1.00	77%	0.23
Design	1.25	85%	0.19
Codierung	1.75	95%	0.09
Sonstige	1.00	86%	0.24
Insgesamt	5.00	85%	0.75



Arten von Anforderungen:

- **funktionale Anforderungen**:

  beschreiben die (primären) Funktionen (Features) des Software-Systems bzw. das Ein-/Ausgabeverhalten des Systems

- **nichtfunktionale Anforderungen**:
  - ⇨ Produktanforderungen: Benutzbarkeitsanforderungen, Effizienzanforderungen, Zuverlässigkeitsanforderungen, Portierbarkeitsanforderungen
  - ⇨ Unternehmensanforderungen: Lieferanforderungen, Umsetzungsanforderungen, Vorgehensanforderungen
  - ⇨ Externe Anforderungen: Kompatibilitätsanforderungen, Ethische Anforderungen, Rechtliche Anforderungen (Datenschutz, Sicherheit)

- … (es gibt noch andere Arten von Anforderungen, aber die obige Einteilung ist die am weitesten verbreitete)



Taxonomy nichtfunktionaler Anforderungen:

[aus EiSE-Skript Michael Eichberg]



3.2 Methoden zur Ermittlung von Anforderungen

Bereits mehrfach wurde auf die Schwierigkeit der Ermittlung der (aller?) Anforderungen an ein (Software-)Produkt hingewiesen. Trotzdem wurde bislang nicht erklärt, wie man Anforderungen ermittelt.

In [WW00] werden u.a. folgende Techniken zur Ermittlung von Anforderungen (**requirements elicitation**) vorgeschlagen:

- Bestimmung aller Interessentengruppen (**stakeholders**) am Software-Produkt und Durchführung von Interviews mit diesen Interessenten
- Ermittlung von Anforderungen aus verschiedenen Sichten (**Viewpoints**); Interessentengruppe legen u.a. Sichten auf ein Software-Produkt fest
- Durchführung von Workshops mit **Brainstorming**-Prozess und/oder **Storyboard**-Techniken, bei denen alle Interessentengruppen vertreten sind
- Ermittlung von **Anforderungsfällen** (typischen Funktionabläufen) und ggf. Durchspielen dieser Fälle mit verteilten Rollen



Stakeholders - Interessengruppen eines Software-Produkts:

Von dem zu erstellenden Software-Produkt sind im Allgemeinen verschiedene Personengruppen mit oft sehr unterschiedlichen (ggf. auch sich widersprechenden) Interessen betroffen. In unserem Fall etwa:

- der Besitzer der Autoverleihfirma (Auftraggeber und Anwender)
- die Aushilfs- bzw. Bürokräfte (Anwender)
- die Kunden der Firma (indirekte Anwender)
- an Entwicklung beteiligte Personengruppen (siehe „Rollen“ in Abschnitt 2.3)
- Betrüger, Diebe, … (Anwender, die das Produkt missbrauchen wollen)

Es ist die erste Aufgabe bei der Erhebung von Anforderungen, alle beteiligten Stakeholders zu identifizieren.

Bei Konflikten zwischen verschiedenen Stakeholder-Gruppen ist es die Aufgabe des Software-Entwicklers, diese zu „schlichten“ (moderieren, aufzulösen, …).



Interviews mit verschiedenen Interessentengruppen:

Durch Interviews (Fragen vorher vorbereiten) oder Fragebogenaktionen wird Bestandsaufnahme = **Domain-Analyse** der Umgebung, in der Software-Produkt eingesetzt werden soll, durchgeführt.

Man versucht Antworten auf etwa folgende Fragestellungen zu erhalten:

- Profil des Kunden (Aufgaben, Produkte, Erfolgsmaße, …)
- Probleme des Kunden (welche Abläufe sind ineffizient, fehlerträchtig, wie werden die beschriebenen Probleme bisher gelöst, wie in Zukunft)
- Umgebung des Kunden (welche Benutzer, was für Hardware-Plattform, …)
- erwartete Eigenschaften des Software-Produktes (Funktionen, Zuverlässigkeit, Support, Installation, …)
- …



Viewpoint-getriebenes Requirements Engineering:

- durch Interessentengruppen definierte Sichten auf ein (Software-)System sind ein effektives Hilfsmittel, den Prozess der Anforderungsanalyse zu organisieren
- andere mögliche „Arten" von Sichten (Viewpoints) sind:
  - ⇨ Orientierung an Funktionsgruppen (Registrierung von Fahrzeugen, …)
  - ⇨ Orientierung an Phasen oder Notationen der Software-Entwicklung

Beispiel für Phasenorientierte Sichten (Philips Research):

1. **Customer View**: Geschäftsmodelle, Marktsituation, … des Kunden
2. **Application View**: Anforderungen aus Sicht einzelner Interessentengruppen
3. **Functional View**: präzise Beschreibung der gewünschten Funktionen
4. **Conceptual View**: (Einschränkungen für) das Design der Software
5. **Realization View**: (Einschränkungen für) die Realisierung der Software



Brainstorming-Workshop zur Ermittlung von Anforderungen:

Alle von dem Software-Produkt betroffenen Personengruppen (beim Auftraggeber und beim Auftragnehmer) werden zu einem Workshop für ein bis zwei Tage zusammengerufen. Der Workshop hat folgende Phasen:

- vorab wird **Material** zur Vorbereitung/Einstimmung **verschickt**
- zunächst werden **Ideen** (für Anforderungen = Funktionen) **gesammelt**
  - ⇨ Ideen auf Zettel an Wand geheftet
  - ⇨ Kritik an geäusserten Ideen verboten
  - ⇨ „wilde" Ideen ausdrücklich erwünscht
  - ⇨ Ergänzungen, Kombinationen von Ideen erwünscht
- dann werden **Ideen gruppiert** und irrelevante Ideen aussortiert und so die gewünschten Funktionen des Software-Produkts bestimmt
- schließlich werden die ermittelten Funktionen nach **Prioritäten** sortiert (durch Abstimmungsprozess beteiligter Stakeholders)



Storyboard-Techniken, Anwendungsfälle und Rollenspiele:

Allen diesen Techniken liegt die Idee zugrunde, dass man die wichtigsten Funktionsabläufe des Software-Produkts durchspielt (Provokation von „**yes, but …** “)

- **Anwendungsfälle** (Use Cases & Misuse Cases):
  - ⇨ alle Funktionen (Features) des Systems werden als exemplarische Abläufe beschrieben (erwünschte und unerwünschte Anwendungsfälle)
  - ⇨ zentrales Element von UML für Anforderungsanalyse (siehe Kapitel 5)
- **Storyboarding**:
  - ⇨ wurde in der Filmindustrie zum ersten Mal für die Produktion von „Schneewittchen und die sieben Zwerge eingesetzt“
  - ⇨ auf einer großen Tafel werden (mit grob skizzierter Benutzeroberfläche) Anwendungsfälle durchgespielt (wie „Fahrzeugreservierung“)
- **Rollenspiele**:
  - ⇨ verschiedene Personen übernehmen die Rollen unterschiedlicher Teile des Software-Systems oder von Benutzern und spielen Abläufe durch



Überprüfung von Anforderungen:

- **Validität**: haben wir die richtigen Anforderungen aufgestellt (ist die beschriebene System-Funktionalität die benötigte)
- **Konsistenz**: sind die beschriebenen Anforderungen widerspruchsfrei (oder kann ein System mit diesen Anforderungen nicht realisiert werden)
- **Vollständigkeit**: fehlen Beschreibungen von Anforderungen für (wichtige) Funktionen (oder Beschreibungen von Teilaspekten einzelner Funktionen)
- **Machbarkeitsprüfung**: kann ein System, das die beschriebenen Anforderungen erfüllt, in einem realistischen Zeitraum mit vetretbaren Kosten realisiert werden
- **Überprüfbarkeit**: kann man die beschriebene Anforderung (durch einen Abnahmetest bei der Auslieferung des Systems) überprüfen
- **Verfolgbarkeit**: sind die Gründe/Motivation für die Aufnahme einer bestimmten Anforderung bekannt, stichhaltig und dokumentiert



3.3 Aufbau und Funktion eines Lastenheftes

Ein Lastenheft für ein Software-Produkt ist nach [Ba96] wie folgt aufgebaut:

1. **Zielbestimmung**: welche Ziele sollen mit dem Software-Produkt erreicht werden.
2. **Produkteinsatz**: Anwendungsbereiche und Stakeholders werden genannt.
3. **Produktfunktionen**: Hauptfunktionen werden beschrieben, Stakeholdergruppen zugeordnet und in 10er-Schritten durchnummeriert (LF nn).
4. **Produktdaten**: permanent gespeicherte Hauptdaten werden festgelegt und in 10er-Schritten durchnummeriert (LD nn).
5. **Produktleistungen**: besondere Anforderungen an Hauptfunktionen oder Hauptdaten (Ausführungszeit, Datenumfang, …) werden aufgezählt (LL nn).
6. **Qualitätsanforderungen**: allgemeine Eigenschaften wie gute Zuverlässigkeit, hervorragende Benutzbarkeit, normale Effizienz, … werden festgelegt.
7. **Ergänzungen**: alles was nicht in obiges Schema passt und trotzdem wichtig ist.
8. **Glossar**: alle in Punkt 1 bis 7 verwendeten wichtigen Begriffe werden erläutert.



Sonstige Eigenschaften des Lastenheftes:

- **Adressaten**: Auftraggeber und Auftragnehmer (Projektleiter, …)
- **Form**: übersichtliche Gliederung, prägnante Sätze in natürlicher Sprache
- **Inhalt**: fundamentale Eigenschaften des Produktes aus Kundensicht
- **Erstellungszeitpunkt**: vor Abschluss eines Vertrages (ggf. als Grundlage dafür)
- **Umfang**: wenige Seiten

Kategorisierung der Funktionen, Daten und Leistungen:

Bei sämtlichen Funktionen, Daten und Leistungen (Anforderungen, Requirements), die im Lastenheft aufgeführt sind, ist ihre Wichtigkeit anzugeben. Üblicherweise wird wie folgt unterteilt:

- absolut notwendig (high priority, primary, …)
- ziemlich wichtig (medium priority, secondary, …)
- Schnickschnack (low priority, optional, frill, …)



Weitere übliche Attribute für Anforderungen:

- **Status**: spiegelt den „Zustand“ einer aufgestellten Anforderung wider (z.B. mit Werten vorgeschlagen, abgesegnet, umgesetzt, …)
- **Nutzen**: erwarteter Nutzen einer Anforderung, der oft mit der Priorität gleichgesetzt wird (z.B. mit Werten kritisch, wichtig, nützlich, …)
- **Aufwand**: geschätzter Aufwand für die Realisierung einer Anforderung (z.B. mit Werten hoch, mittel, niedrig, …)
- **Risiko**: Schätzung der Wahrscheinlichkeit, dass es bei Realisierung der Anforderung zu (un-)erwarteten, unerwünschten Problemen kommt, die Projekt gefährden
- **Stabilität**: Schätzung der Wahrscheinlichkeit, dass sich diese Anforderung im Projektverlauf noch ändern wird (z.B. mit Werten hoch, mittel, niedrig, …)
- **Release**: Angabe des Releases (Produktgeneration), die die Anforderungen realisieren soll
- **Grund**: Verweis auf eine Quelle, die begründet, warum die Anforderung aufgestellt wurde bzw. welcher konkreter Nutzen mit ihrer Realisierung verbunden ist



Beispiel “Kundenbeschreibung eines Software-Produkts”:

Motor Vehicle Reservation System (MVRS)

A rental office lends motor vehicles of different types. The assortment comprises cars, vans, and trucks. Vans are small trucks, which may be used with the same driving license as cars. Some client may reserve motor vehicles of a certain category for a certain period. He or she has to sign a reservation contract. The rental office guarantees that a motor vehicle of the desired category will be available for the requested period. The client may cancel the reservation at any time. When the client fetches the motor vehicle he or she has to sign a rental contract and optionally an associated insurance contract. Within the reserved period, at latest at its end, the client returns the motor vehicle and pays the bill.



Probleme mit der Kundenbeschreibung:

- sie enthält keine Aussagen darüber, **warum** der Kunde genau den skizzierten Geschäftsprozess durch Software unterstützen will (z.B. weil Reservierungen bereits öfters vergessen wurden, Wartungsintervalle überschritten wurden, …)
- sie enthält **undefinierte Begriffe**, unter denen Auftragnehmer und Auftraggeber sich möglicherweise unterschiedliches vorstellen (z.B. *assortment*, *category*, …)
- sie enthält möglicherweise **irrelevante Aussagen** (z.B. „*Vans are small trucks, which may be used with the same driving licence as cars*“)
- sie ist noch sehr **unvollständig** (z.B. ist unklar was passiert, wenn Fahrzeuge nicht rechtzeitig zurückgegeben werden, Fahrzeuge trotz Reservierung nicht abgeholt werden, …)
- es ist unklar, welche Teile des beschriebenen Geschäftsprozesses durch Software unterstützt werden sollen (**Festlegung der Systemgrenze** fehlt)
- sie enthält ausschließlich Aussagen über den **funktionalen Ablauf** des betrachteten Geschäftsprozesses (keine Angaben zu Datenmengen, Antwortzeiten, Art der Benutzer, …)



Klassisches Beispiel für Mehrdeutigkeit von Aussagen:

Untersuchung des Satzes „*Mary had a little lamb*“ durch Hervorhebung einzelner Wörter und Nachschlagen der Wörter in einem Wörterbuch (Glossar):

- **have**:
  1) to hold in possession as property
  2) to trick or fool someone (been had by a partner)
  3) to beget or bear (have a baby)
  4) to partake (have as dinner)
  5) …

- **lamb**:
  1) a young sheep less than one year without permanent teeth …
  2) the young of various other animals (antelope etc.)
  3) a person as gentle or weak as a lamb …
  4) a person easily cheated or deceived
  5) the flesh of lamb used as food
  6) …



Mögliche Bedeutungen von „Mary had a little lamb":

have lamb	Bedeutung
1	1 *Mary owned a little sheep under one year …*
2	4 *Mary tricked a person easily cheated …*
3	2 *Mary gave birth to an antelope*
4	5 *Mary ate a little of the lamb*
…	

Konsequenzen für die Erhebung von Anforderungen:

- Aus einer natürlichsprachlichen (informellen) Anforderungsdefinition alle vermutlich wichtigen (bedeutungstragenden) Wörter heraussuchen
- für alle wichtigen und/oder möglicherweise unklaren Begriffe ein **Glossar** anlegen, das Lastenheft (und später Pflichtenheft) ergänzt
- mehr Informationen zu Techniken für präzisere (semiformale) Beschreibung von Anforderungen mit Hilfe von UML in Kapitel 5



Zurück zum Lastenheft des Software-Produkts:

1. **Zielbestimmung**:

   Das Programm MVRS soll eine kleine Autovermietung mit genau einer Niederlassung in die Lage versetzen, die Buchung und Verleihung ihrer verschiedenen Wagentypen (Arten, Kategorien) zu verwalten.

2. **Produkteinsatz**:

   Das Produkt wird im Verkaufsraum und im Büro der Firma vom Besitzer der Firma und oft wechselnden Aushilfskräften bedient. …

3. **Produktfunktionen**:

   /LF10/ Ersterfassung, Änderung und Löschung von Fahrzeugen.

   /LF20/ Ersterfassung, … von Fahrzeugtypen.

   /LF30/ Ersterfassung, … von Fahrzeugreservierungen.

   /LF40/ Ausgabe verfügbarer Fahrzeuge eines Typs in einem bestimmten Zeitintervall mit folgenden Daten: … .

   /LF50/ …



4. **Produktdaten**:

   /LD10/ Folgende Daten sind zu jedem Fahrzeug zu speichern: Typ, Farbe, gefahrene Kilometer, letzte Inspektion, … .

   /LD20/ Folgende Daten sind bei jeder Fahrzeugreservierung zu speichern: Zeitraum, gewünschter Typ (vorreserviertes Fahrzeug), Adresse des Kunden

   /LD30/ …

5. **Produktleistungen**:

   /LL10/ Bei der Listenausgabe der Funktionen /LF40/, … werden zunächst nur die ersten n “Treffer” ausgegeben, weitere Treffer nur auf Wunsch.

   /LL20/ Das System erzwingt die regelmäßige Erstellung von Datensicherungen für die Daten /LD20/, … .

   /LL30/ 100 Fahrzeuge und 1000 Reservierungen werden maximal gespeichert.

   /LF30/ Die Bearbeitung einer Fahrzeugreservierung dauert nicht länger als 10 Sekunden.



6. **Qualitätsanforderungen**:

Produktqualität	sehr gut	gut	normal	irrelevant
Funktionalität		**x**		
Zuverlässigkeit	**x**			
Benutzbarkeit	**x**	**x**		
Effizienz			**x**	
Änderbarkeit			**x**	
Portierbarkeit				**x**

Die Benutzbarkeit der Funktionen /LF10/ und /LF20/ muss gut sein, da sie allein vom Besitzer der Firma verwendet werden. Die Benutzbarkeit aller übrigen Funktionen muss sehr gut sein, da sie von wechselnden Aushilfskräften bedient werden.

7. **Ergänzungen (wie z.B. Abgrenzungskriterien)**:

Die Zuordnung verfügbarer Fahrzeuge zu Reservierungen erfolgt in der ersten Version manuell, Buchhaltungsfunktionen gehören nicht zum Leistungsumfang.



Glossar zu Lastenheft:

Das Glossar ist entweder Bestandteil des Lastenheftes oder auch ein separates Dokument (da es später weiterverwendet wird). Es erläutert alle wichtigen Begriffe der Welt der Anwender (Stakeholders) knapp aber präzise.

Beispiele für Begriffe:

- **Fahrzeug**: ein Automobil, das entweder mit Führerschein Klasse 3 oder Führer-schein Klasse 2 gefahren werden darf (Fahrräder, Motorräder, Schiffe, … werden nicht betrachtet).
- **Fahrzeugtyp**: ein bestimmtes Fabrikat eines Herstellers wie Smart, Corsa, … .
- **Aushilfskraft**: für einen beschränkten Zeitraum und maximal 12 Stunden pro Woche in der Autovermittlung am Kundenschalter arbeitende Person.
- …



Erstellung von Lastenheft mit Process (otris Software AG, Dortmund):

PROCESS - Pflichtenheftgenerator

Anwendung Stammdatenlisten Ersterfassung Administration Hilfe

Neu Bearbeite Aktualisiere Löschen Anpasser

Projekt (2 Einträge)
MVRS - Motor Vehicle Re
Glossare (1 Eintrag)
Version 0.9 vom 1
Pflichtenhefte (1 Eintr.
Version 0.9 vom 1
Lastenhefte (1 Eintrag
Version 0.9 vom 1
Seminarorganisation
Glossare (1 Eintrag)
Pflichtenhefte (1 Eintr.
Lastenhefte (1 Eintrag
Vorlage (0 Einträge)
Historie
Favoriten (0 Einträge)

Lastenheft MVRS - Motor Vehicle Reservation System
Version 0.9 vom 10.04.2005

Drucken

Version **0.9**
Autor **Andy Schürr**
Datum **10.04.2005**
Status **in Arbeit**
Kommentar

Zielbestimmung

Das Programm MVRS soll eine kleine Autovermietung mit genau einer Niederlassung in die Lage versetzen, die Buchung und Verleihung ihrer verschiedenen Wagentypen (Arten, Kategorien) zu verwalten.

Produkteinsatz

Das Produkt wird im Verkaufsraum und im Büro der Firma vom Besitzer der Firma und oft wechselnden Aushilfskräften bedient. ...

Übersicht:

Produktfunktionen:

/Nr./	Gesch
10	Ersterfassung, Änderung und Löschung von Fahrzeugen
20	Ersterfassung, ... von Fahrzeugtypen
30	Ersterfassung, ... von Fahrzeugreservierungen

Produktdaten:

/Nr./	Bezeic
10	Folgende Daten sind zu jedem Fahrzeug zu speichern: Typ, Farbe, g
20	Folgende Daten sind bei jeder Fahrzeugreservierung zu speichern: Z Kunden

Produktleistungen:

/Nr./	Bezeic
10	Bei der Listenausgabe der Funktionen /LF40/, ... werden zunächst n
20	Das System erzwingt die regelmäßige Erstellung von Datensicherung

Qualität:
Funktionalität
Benutzbarkeit
Zuverlässigkeit
Effizienz
Änderbarkeit
Übertragbarkeit

Lastenheft-Version 0.9 vom 10.04.2005

Autor: Andy Schürr
Datum: 10.04.2005 Version: 0.9
Status: in Arbeit / akzeptiert / freigegeben
Kommentar

1.Ziel | 2.Einsatz | 3.Übersicht | 4. Funktionen | 5. Daten | 6. Leistungen | 7. Qualität | 8. Ergänzungen

Zielbestimmung

Das Programm MVRS soll eine kleine Autovermietung mit genau einer Niederlassung in die Lage versetzen, die Buchung und Verleihung ihrer verschiedenen Wagentypen (Arten, Kategorien) zu verwalten.

OK | Übernehmen | Abbrechen | zurück | weiter | FP-Bewertung | Drucken



Erstellung von Glossar mit Process (otris Software AG, Dortmund):

PROCESS - Pflichtenheftgenerator

Anwendung Stammdatenlisten Ersterfassung Administration Hilfe

Neu Bearbeite Aktualisiere Löschen Anpasser

- Projekt (2 Einträge)
  - MVRS - Motor Vehicle Re
    - Glossare (1 Eintrag)
      - Version 0.9 vom 1
    - Pflichtenhefte (1 Eintr.
      - Version 0.9 vom 1
    - Lastenhefte (1 Eintrag
      - Version 0.9 vom 1
  - Seminarorganisation
    - Glossare (1 Eintrag)
    - Pflichtenhefte (1 Eintr.
    - Lastenhefte (1 Eintrag
- Vorlage (0 Einträge)
- Historie
- Favoriten (0 Einträge)

Glossar MVRS - Motor Vehicle Reservation System
Version 0.9 vom 10.04.2005

Drucken

Version	**0.9**
Autor	**Andy Schürr**
Datum	**10.04.2005**
Status	**in Arbeit**
Kommentar	

Begriffe:

Bezeichnung	Erklärung
Autovermietung	Eine selbständige Firma, die genau eine Niederlassung besitzt und eine kleiner Anzahl von Autos verleiht.
Fahrzeug	Ein konkretes Fahrzeug, das der Autovermietung gehört.
MVRS	Abkürzung für Motor Vehicle Reservation System; der Name des zu entwickelnden Softwaresystems.

Glossar-Version 0.9 vom 10.04.2005

Autor: Andy Schürr
Datum: 10.04.2005 Version: 0.9
Status: (•) in Arbeit ( ) akzeptiert ( ) freigegeben
Kommentar

Einträge

Bezeichnung	Erklärung
Autovermietung	Eine selbständige Firma, die genau eine Niederlass...
Fahrzeug	Ein konkretes Fahrzeug, das der Autovermietung g...
MVRS	Abkürzung für Motor Vehicle Reservation System; d...

OK Übernehmen Abbrechen Drucken



3.4 Aufwands- und Kostenschätzung

Die **Kosten** eines Software-Produktes und die **Entwicklungsdauer** werden im wesentlichen durch den personellen Aufwand bestimmt. Letzterer ergibt sich aus

- ⇨ dem “**Umfang**” des zu erstellenden Software-Produkts
- ⇨ der geforderten **Qualität** für das Produkt

Übliches Maß für Personalaufwand:

Mitarbeitermonate (MM) oder Mitarbeiterjahre (MJ):

1 MJ ≈ 10 MM (wegen Urlaub, Krankheit, …)

Übliche Maße für Produktumfang:

- ⇨ „Lines of Code” (LOC) = Anzahl Code-Zeilen ohne Kommentare
- ⇨ „Function Points“ (FP) = Anzahl von Einzelfunktionen

Bestimmung der Produktqualität:

siehe Abschnitt 1.4 und Kapitel 10



Schätzverfahren im Überblick:

- **Analogiemethode**: Experte vergleicht neues Projekt mit bereits abgeschlossenen ähnlichen Projekten und schätzt Kosten “gefühlsmäßig” ab
  - ⇨ Expertenwissen lässt sich schwer vermitteln und nachvollziehen
- **Prozentsatzmethode**: aus abgeschlossenen (ähnlichen) Projekten wird Verteilung des Aufwands auf Phasen ermittelt. Anhand abgeschlossener Phasen wird Restlaufzeit des Projekts geschätzt
  - ⇨ funktioniert allenfalls nach Abschluss der Analysephase
- **Parkinsons Gesetz**: die Arbeit ist beendet, wenn alle Vorräte aufgebraucht sind.
  - ⇨ praxisnah, realistisch und wenig hilfreich …
- **Gewichtungsmethode**: Bestimmung vieler Faktoren (Erfahrung der Mitarbeiter, verwendete Sprachen, …) und Verknüpfung durch mathematische Formel
  - ⇨ LOC-basierter Vertreter: COnstructive COst MOdel (COCOMO)
  - ⇨ FP-basierte Vertreter: Function-Point-Methoden in vielen Varianten



3.5 Projektpläne und Projektorganisation

Am Ende der Machbarkeitsstudie steht die Erstellung eines Projektplans mit

- ⇨ Identifikation der einzelnen **Arbeitspakete**
- ⇨ **Terminplanung** (zeitliche Aufeinanderfolge der Pakete)
- ⇨ **Ressourcenplanung** (Zuordnung von Personen zu Paketen, …)

Hier wird am deutlichsten, dass eine Machbarkeitsstudie ohne ein grobes Design der zu erstellenden Software nicht durchführbar ist, da:

- ⇨ Arbeitspakete ergeben sich aus der Struktur der Software
- ⇨ Abhängigkeiten und Umfang der Pakete ebenso
- ⇨ Realisierungsart der Pakete bestimmt benötigte Ressourcen

Konsequenz: Projektplanung und -organisation ist ein fortlaufender Prozess. Zu Projektbeginn hat man nur einen groben Plan, der sukzessive verfeinert wird (siehe Kapitel 10).



3.6 Eine abschließende Checkliste

Eine Machbarkeitstudie sollte (in der Regel) nur wenige Tage dauern. Gegebenenfalls ist ein Vorprojekt sinnvoll, in dem technische Risiken, etc. studiert werden. Es müssen folgende Punkte geklärt sein:

- ❑ Sind die **Angaben des Lastenheftes** mit allen maßgeblichen Parteien auf Kundenseite abgestimmt (Vorsicht mit Interessensgegensätzen auf Kundenseite)?
- ❑ Ist das Produkt mit den eingeplanten Ressourcen **technisch realisierbar**?
- ❑ Gibt es eine (hinreichend vertrauenswürdige) **Kostenschätzung**?
- ❑ Ist Produktentwicklung **wirtschaftlich** interessant (oder "politisch" notwendig)?
- ❑ Ist ein (realistischer) **Zeitplan** vorhanden mit Datumsangaben für alle Phasenübergänge (Datumsangaben für Einzelaktivitäten können noch fehlen)?
- ❑ Gibt es für alle Phasen einen **Phasenverantwortlichen** (Zuteilung von Ressourcen zu Einzelaktivitäten kann noch fehlen)?
- ❑ Wurde **Risikoabschätzung** durchgeführt (Ausfall v. Gruppenmitgliedern, …)?



3.7 Weitere Literatur

[Al98] A.J. Albrecht: *Measuring Application Development Productivity*, in Guide, Share (eds.): Proc. of th eIBM Applications Development Symposium, Monterey, Ca. (1979), 83-92

Die Originalquelle für die Function-Point-Methode zur Kostenschätzung.

[Ba96] H. Balzert: *Lehrbuch der Software-Technik (Band 1): Software-Entwicklung*, Spektrum Akademischer Verlag (1996)

Der Aufbau von Lastenheften und die Darstellung der Function-Point-Methode zur Kostenschätzung wurde aus diesem Buch für die Vorlesung übernommen.

[De98] T. DeMarco: *Der Termin - Ein Roman über Projektmanagement*, Hanser-Verlag (1998), 266 Seiten

Mein Lieblingsbuch für eine Einführung in die Probleme des Managements von Software-Projekten. In der Form eines Krimis geschrieben (Hauptdarsteller ist ein Methodenberater, der in osteuropäisches Land zur Sanierung der dortigen Software-Industrie entführt wird). Als Abendlektüre sehr empfehlenswert!

[SZ98] G. Snelting, A. Zeller: *Einführung in Software Engineering*, Folien einer Vorlesung an der TU Braunschweig

Dieser Vorlesung wurden die Regeln für die Erstellung lesbarer Dokumente entnommen.

[WW00] D.L. Well, D. Widdrig: *Managing Software Requirements - A Unified Approach*, Addison Wesley (2000), 491 Seiten

Diesem Buch wurden einige Beispiele und Techniken zur Ermittlung von Anforderungen an ein Software-System entnommen.



4. Grundlagen der objektorientierten Modellierung

Themen dieses Kapitels:

- strukturierte versus objektorientierte Softwaremodellierung
- Grundbegriffe der Objektorientierung (Klasse, Vererbung, …)
- Geschichte der Standard-OO-Modellierungssprache UML

Achtung:

Dieses Kapitel ist **keine** Einführung in die objektorientierte Programmierung (dafür gibt es ein Praktikum und eigene Vorlesungen). Hier werden „nur” die Grundideen der objektorientierten Softwareentwicklung zusammengefasst.



Mittel der Softwaretechnik - der „Regenbogen“:



4.1 Modellierung von Software-Systemen

Problemgebiet (Anforderungen)

Objekte und Abläufe im Umfeld einer Autovermietung

korrekte Umsetzung ?

Implementierung des Motor Vehicle Reservation Systems

Programm



Vom Problem zum Programm:

Problemgebiet

Objekte und Abläufe im Umfeld einer Autovermietung

korrekt ?

Analysemodell

was für Objekte und Abläufe sind für das Programm relevant und wie sehen sie **abstrahiert** aus

welche Objekte und Abläufe werden im Programm realisiert

korrekt ?

wie werden die Objekte im Programm dargestellt

und wie werden die Abläufe durch Funktionen realisiert (unterstützt)

Entwurfsmodell

korrekt ?

Implementierung des Motor Vehicle Reservation Systems

Programm

Achtung:

Abläufe werden durch das **Verhalten** miteinander kooperierender Objekte festgelegt.



Strukturierte Softwareentwicklungsmethode der 80/90er Jahre:



Eingesetzte klassische Modellierungssprachen:

- Für die Modellierung funktionaler Aspekte (Verhalten):
  - **Datenflussdiagramme** = durch Kanäle verbundene Prozessoren
  - **Funktionsbäume** = hierarchische Dekomposition von Algorithmen
- Für die Modellierung von Datenstrukturen:
  - **Data Dictionary** = Ansammlung von Typdefinitionen
  - **Entity-Relationship-Diagramme** (Klassendiagramme ohne Operationen)
- Für die Modellierung von Kontrollstrukturen:
  - **Struktogramme** (Nassi-Shneiderman-Diagramme)
  - **Pseudocode**
- Für die Modellierung zustandsbehafteter (nebenläufiger) Systeme:
  - **Zustandsübergangsdiagramme** (endliche Automaten)
  - **Petrinetze**

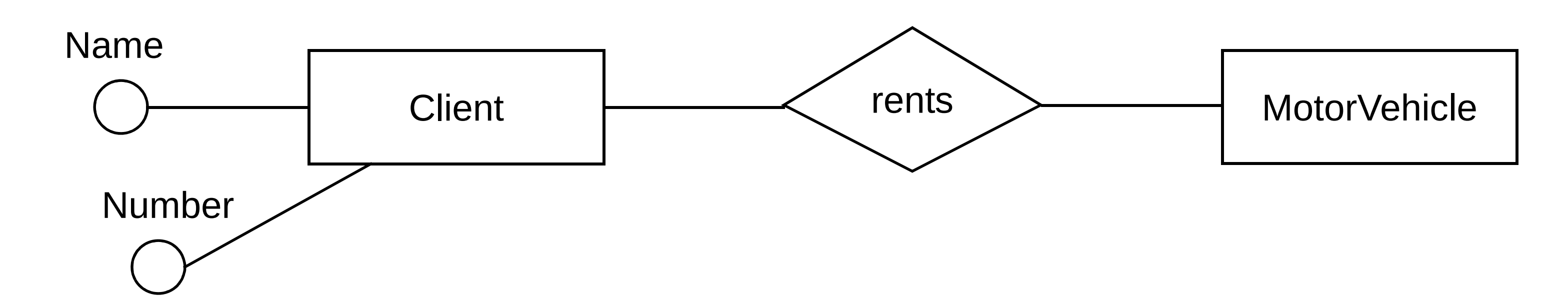


Beispiel eines einfachen Entity-Relationship-Diagramms (ER-Diagramm):



Erläuterungen zu ER-Diagrammen:

- ER-Diagramme werden vor allem für den Datenbank-Entwurf eingesetzt
- Entities entsprechen Klassen einer OO-Sprache (ohne Methoden)
- Einfache Relationships entsprechen Zeigerpaaren
- Attribute entsprechen Attributen …




```
public class Client {
    private int number;
    private String name;}
```

```
public class Motorvehicle {
    private Client lnkRevRents;

    private Motorvehicle lnkRents;
}
```



Beispiel eines „einfachen“ Datenflussdiagramms:

reserveVehicle:

Client Name
Reservation Period
ErrorMessage
(Reservierung hat nicht geklappt)
getClientNo
ClNo
Name
ClNo
Client-DB
VNoSet
selectVehicle
Vehicle Category
VNoSet Period
VNo
VNo ClNo Period
findVehicles
Contract-DB
Category
VNoSet
Period VNo, ClNo
ResNo
Vehicle-DB
mkReservation
ResNo

Legende:
Prozess
Datenbank
Datenfluss



Erläuterungen von Datenflussdiagrammen:

- sie eignen sich gut für die Beschreibung des Aufbaus eines Systems, das aus Komponenten besteht, die über Verbindungen miteinander kommunizieren
- für die Beschreibung von Berechnungsabläufen ist ihre Bedeutung (Semantik) aber nicht klar genug definiert:
  - ⇨ Müssen alle in einen Prozess einlaufenden Datenflüsse mit Daten belegt sein, bevor der Prozess rechnen kann?
  - ⇨ Liegen auf den Datenflüssen immer Daten an oder werden Signale verschickt, die bei Berechnungen konsumiert werden?
  - ⇨ In welcher Reihenfolge müssen oder dürfen Berechnungen stattfinden und wie oft (wie lange) rechnet ein Prozess?
  - ⇨ Wie wird auf die Datenbanken zugegriffen?
  - ⇨ …



Probleme mit der klassischen Softwareentwicklungsmethode:

- ☹ starke Trennung zwischen Daten- und Verhaltensaspekten
  - ⇨ resultiert in enormen Konsistenzhaltungsproblemen
  - ⇨ ggf. werden nur Daten oder Verhalten modelliert
- ☹ verschiedene Konzepte und Sprachen in den einzelnen Entwicklungsphasen
  - ⇨ resultiert in enormen Konsistenzhaltungsproblemen
  - ⇨ allerdings: (gewünschte) Trennung von Analyse und Entwurf wird erzwungen

Voraussetzung für erfolgreiche Softwareentwicklung:

- ❑ Modellierungssprachen mit wohldefinierter Syntax und Semantik
- ❑ präzise sprachübergreifende Konsistenz- und Abbildungsregeln



Neue Idee des integrierten Modellierungsansatzes:

- enge Integration von Daten- und Verhaltensbeschreibung
- ein Konzept für alle Phasen der Software-Entwicklung

Das Zauberwort:

Objektorientierung



Vision der objektorientierten Softwareentwicklung:

Abkürzungen:

OOA = OO-Analyse
OOD = OO-Design
OOP = OO-Programmierung
OODB = OO-Datenbanken



Realität der objektorientierten Modellierung:



4.2 Grundbegriffe der Objektorientierung

Objektorientierte Programmierung nach [Bo94]:

" … a method of implementation in which programs are organized as **cooperative collections of objects***, each of which represents an* **instance of some class***, whose classes are all members of a* **hierarchy of classes** *united via* **inheritance** *relationships."*

"Essentials" der Objektorientierung sind also:

1. alles ist ein **Objekt**

Objekte sind Instanzen von **Klassen**, die ihr Verhalten festlegen

Objekte **kommunizieren** (über Nachrichtenaustausch)

Klassen bilden eine **Vererbungshierarchie**



Weitere "Essentials" der Objektorientierung:

Objekte besitzen eine unveränderliche **Identität**

Objekte besitzen einen veränderlichen **Zustand**

Objekte besitzen ein dynamisches **Verhalten**

Objekte sind **Abstraktionen** (der realen Welt)

Objekte **verkapseln** ihren internen Zustand

Objekte besitzen einen **polymorphen Typ** (dynamisches Binden)

Objekte können **nebenläufig** aktiv sein

Objekte können **persistent** sein (dauerhaft gespeichert)

Oder nach Roger King:

My cat is object-oriented: *"After all a cat exhibits characteristic behavior, responds to messages, is a heir of a long tradition of inherited responses, and manages its own quite independent internal state."*



Konkrete und abstrakte Objekten:

Problemgebiet

konkrete Objekte (der realen Welt)

Abstraktion von unwesentlichen Details

Analysemodell

abstrakte Objekte (im Modell)

Entwurfsmodell

deskriptive Modelle der realen Welt

präskriptive Modelle der Implementierung

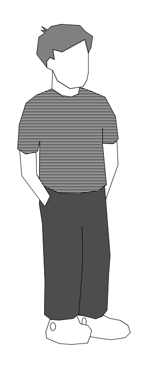
Abstraktion von Implementierungs-details

Programm

konkrete Laufzeitobjekte (im Programm)



Beispiele für konkrete Objekte der realen Welt:



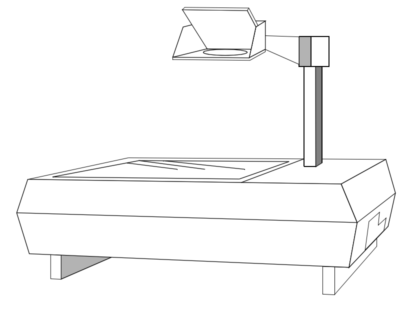
Timo

eine konkrete Person:

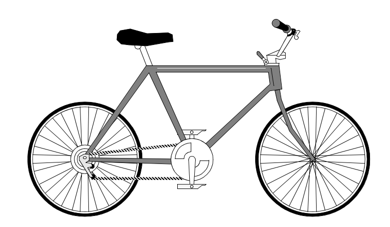
- eindeutig identifizierbar
- hat verschiedene Eigenschaften, die den aktuellen Zustand bestimmen
- verweist ggf. auf andere Objekte
- kennt ein dynamisches Verhalten



Telefon im
Raum E4.324



Projektor in F1.110



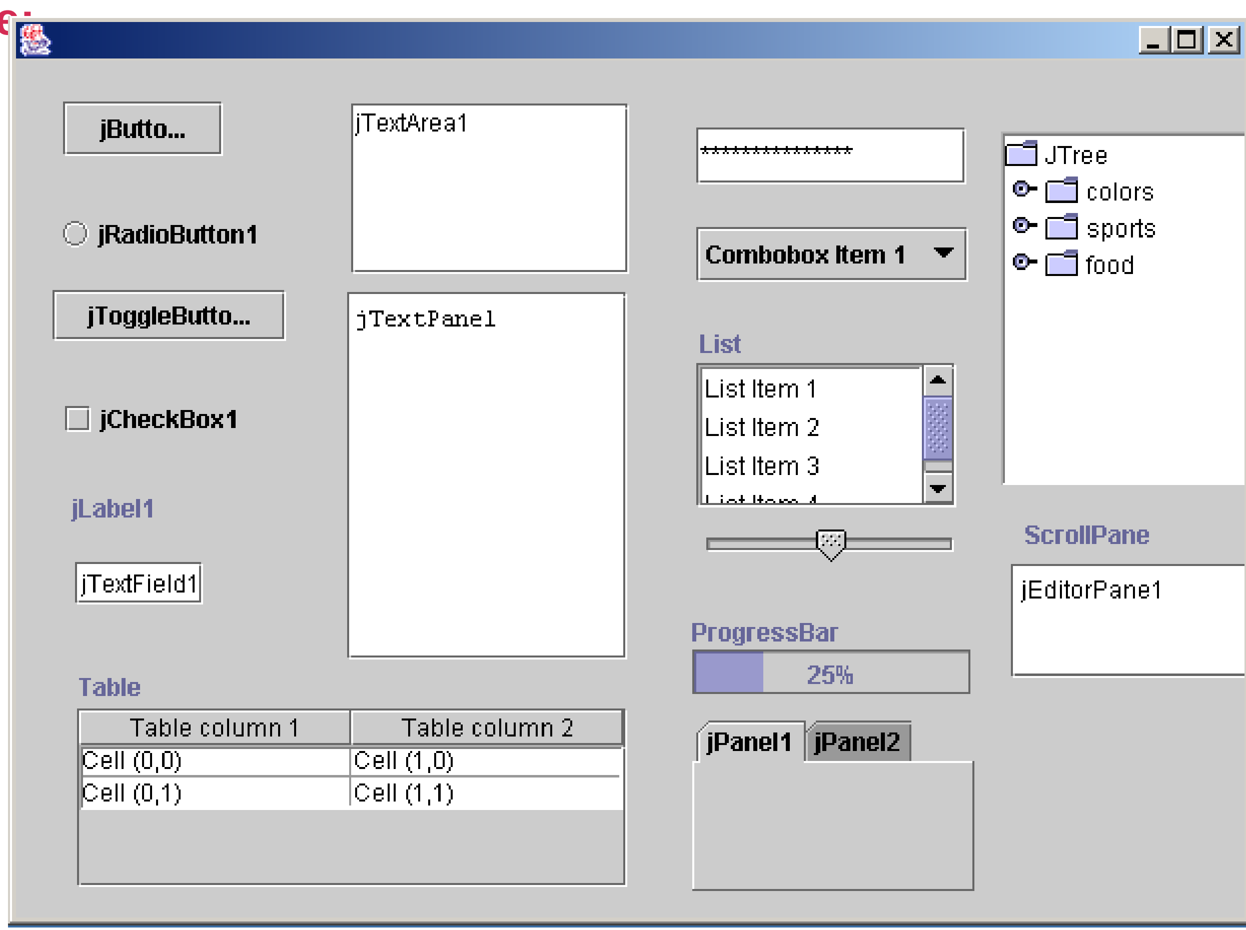
das Fahrrad von Timo



Beispiel für Laufzeitobjekte in Programmen:

Swing-GUI-Elemente:

- Id = Adresse in JVM
- Zustand = Position, …
- Verhalten = …





Objektidentität und Objektgleichheit:

- jedes Objekt ist systemweit eindeutig identifizierbar
- es besitzt einen **Objektbezeichner** (object identifier = oid)
- Objektbezeichner wird
  - ⇨ bei der “Geburt” vergeben
  - ⇨ ist unabhängig vom aktuellen Zustand
  - ⇨ kann nicht verändert werden
- nur der Zustand eines Objekts ist veränderbar



Objektidentität ≠ Objektgleichheit

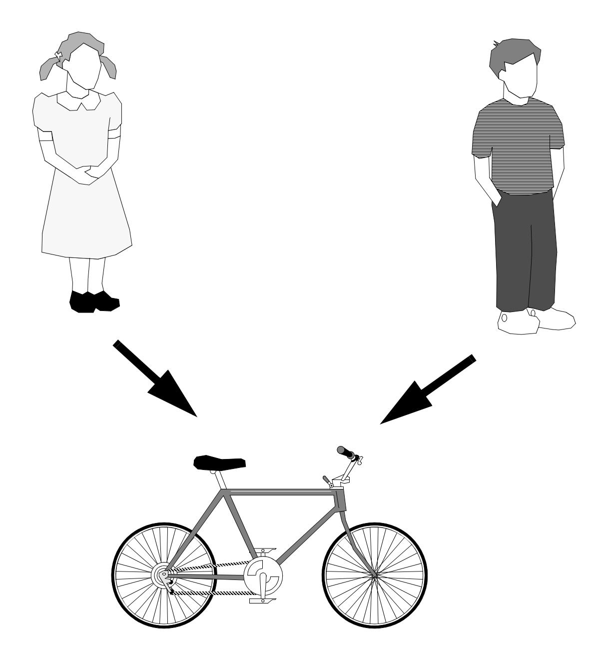


Beispiel für Objektidentität:

"Timo und Hannah haben **ein** *Mountainbike."*

⇨ beide sind Besitzer desselben Fahrrads

⇨ zwei Verweise auf dasselbe Fahrrad

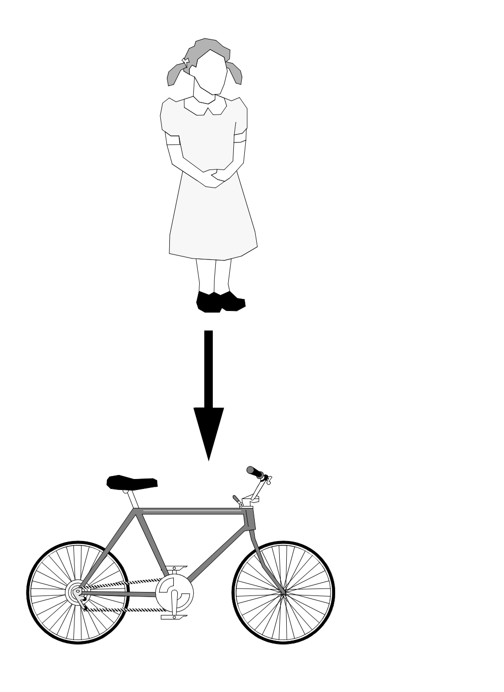


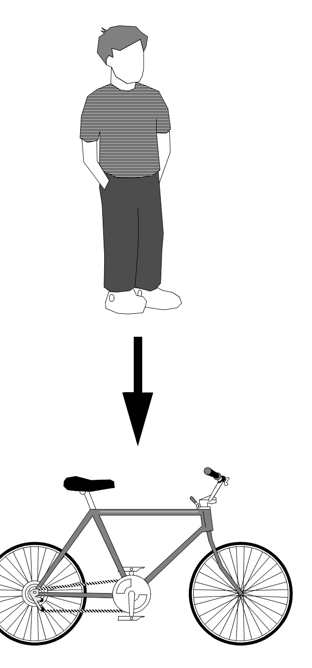


Beispiel für Objektgleichheit:

"Timo und Hannah **haben** *ein Mountainbike."*

- ⇨ beide sind Besitzer eines (eigenen) Fahrrads
- ⇨ zwei Objekte mit dem gleichen internen Zustand

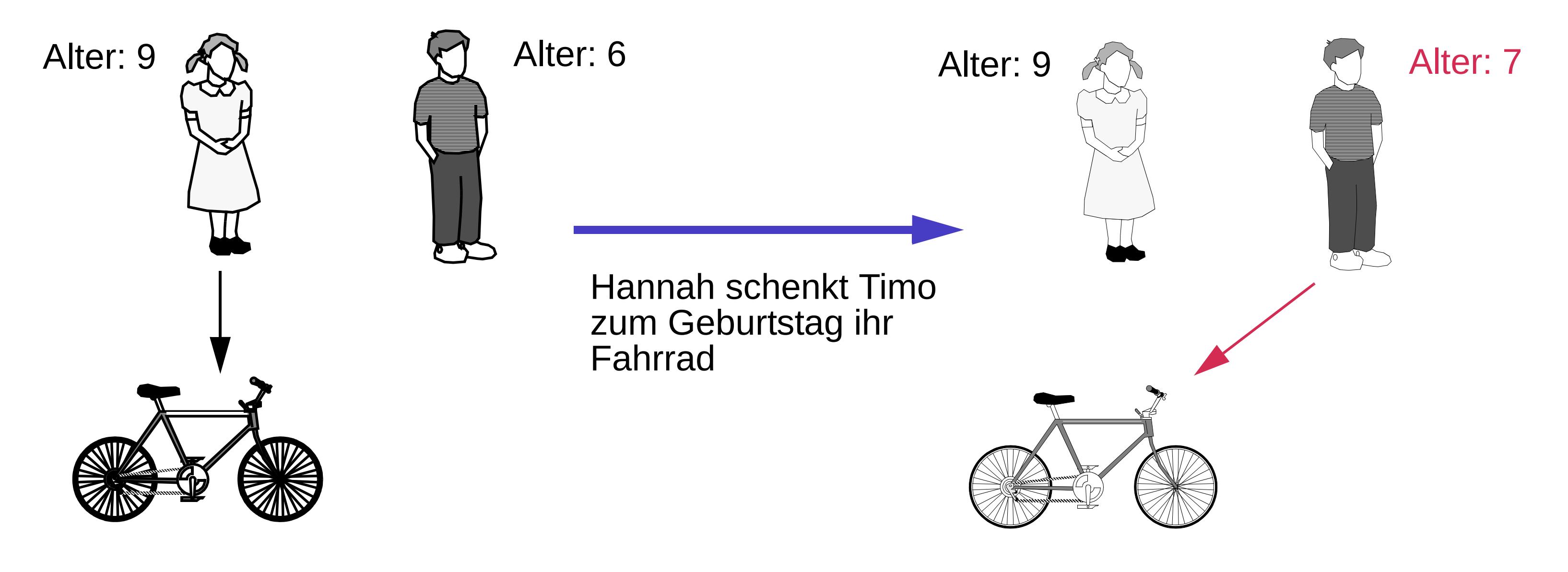






Objektzustand:

- der Zustand eines Objektes wird durch seine **Attributwerte** festgelegt
- Attributwerte können
  - ⇨ "primitive" **Datentypwerte** sein (z.B. int)
  - ⇨ Zeiger auf andere Objekte sein
- der Zustand eines Objektes kann sich ändern:





Objektklassen:

- gleichartige Objekte werden in einer **Klasse** zusammengefasst
- jedes **Objekt** ist Instanz genau einer Klasse
- alle Instanzen einer Klasse sind Bestandteil ihrer **Extension**
- ein Objekt ändert seine Klasse während seiner Lebenszeit nicht
- die Klasse legt für ihre Instanzen fest:
  - ⇨ alle **Attribute** mit zulässigen Wertebereichen
  - ⇨ alle **Operationen** mit ihren Implementierungen als Methoden
  - ⇨ die Trennung von **Schnittstelle** = öffentliche Operationen der Klasse und **Rumpf** = Implementierung dieser Operationen
- Klassen bilden **Kapseln** (abstrakte Datentypen = ADTs) für ihre Objekte
- **abstrakte** Klasse = Klasse ohne Instanzen mit unvollständiger Implementierung



Von konkreten Objekten zu Klassen:

Auto (**Klasse**)
preis : Euro
ort : String
gibPreis() : Euro
bewege(…)

private: interner Zustand

public: Schnittstellenoperatio

Abstraktion

M - V 6120 : Auto
preis = 30.000
ort = “München“

DA - T 4378 : Auto
preis = 20.000
ort = “Darmstadt“

Abstraktion



Klassen in Java realisiert:

Auto (Klasse)
preis : Euro ort : String
gibPreis() : Euro bewege(…)

private: interner Zustand

public: Schnittstellenoperatio

```
public class Auto {

    private Euro preis;
    private String ort;

    public Euro gibPreis( ) {
        return preis; }

}
```



Klassenhierarchien und Vererbung:

- Vererbung ("**inheritance**") ist ein Mechanismus um neue Klassen mit Hilfe bereits bestehender Klassen zu definieren
- damit lassen sich Klassen wiederverwenden ("**reuse**")
- eine Unterklasse ("**subclass**") **erbt** von der Oberklasse ("**superclass**")
  - ⇨ die Schnittstelle(n) mit allen von aussen aufrufbaren Operationen
  - ⇨ die Implementierungen der Operationen (ggf. mit Hilfsmethoden)
  - ⇨ alle Attributdeklarationen (Instanzvariablen)
- eine Unterklasse kann
  - ⇨ ererbte Operationen (und manchmal auch Attribute) **redefinieren**
  - ⇨ **neue** Attribute und Operationen definieren
- durch Vererbung entstehen **Klassenhierarchien**
- Vererbung ist eine **transitive** Beziehung



Vererbungsbeispiel:

Person
Name Alter
erhöheAlter

← Attribute der Klasse

← Operationen der Klasse

△ (Vererbung)

Mitarbeiter
Gehalt
erhöheGehalt

← neue Attribute der Klasse

← neue Operationen der Klasse

Achtung: die ererbten Attribute und Operationen werden in aller Regel nicht wieder aufgeführt.



Polymorphie in (objektorientierten) Programmiersprachen:

Das Wort „**Polymorphie**“ kommt aus dem Griechischen und heisst „vielgestaltig”.
Als polymorph werden alle Programmiersprachen bezeichnet, die

die Definition von Prozeduren (Operationen) erlauben, die auf einer bestimmten Eingabeparameterposition Aktualparameter unterschiedlicher Typen (Klassen) erlauben.

Vorteil:

Statt vieler ähnlicher Prozeduren für jeden Typ hat man nur eine Prozedur, in der man Fehler suchen muss, die man abändern muss, … .

Beispiele polymorpher Programmiersprachen:

- funktionale Sprachen wie Haskell, ML, Miranda, …
- alle objektorientierten Sprachen wie Java, C++, …
- dynamisch (nicht statisch) getypte Sprachen wie Lisp, Smalltalk, …



Beispiel für Polymorphie:

Achtung:

Die Operation gibEinkommen ist polymorph, da sie auf Objekten der Klassen Person, Mitarbeiter und Chef aufgerufen werden kann.



Dynamisches Binden:

- in polymorphen funktionalen Sprachen wird derselbe Programmcode für Eingabe-parameter unterschiedlicher Typen abgearbeitet (kein dynamisches Binden)
- bei objektorientierten Sprachen kann ein bestimmter Operationsaufruf in jeder erbenden Unterklasse durch eine völlig andere Methode implementiert sein
- die klassenabhängige Auswahl einer bestimmten Implementierungsmethode zu einem Operationsaufruf (zu einer empfangenen Nachricht) zur **Programmlaufzeit** nennt man **dynamisches Binden**
- im Gegensatz dazu wird die Auswahl einer bestimmten Prozedur aus einer Menge gleichbenannter Prozeduren zur Übersetzungszeit **Overloading** genannt (anhand der Aktualparametertypen)

  Beispiel: der Operator “+” kann auf integer, cardinal, float, … realisiert sein
  Der Übersetzer entscheidet anhand der Aktualparameter welchen Maschinencode er für das “+” erzeugen muss.



Beispiel für dynamisches Binden (Pseudocode):

```
einePerson = … (* Mitarbeiter oder Chef *);
ihrEinkommen = einePerson.gibEinkommen()
```

Realisierung in imperativer Sprache:

```
einePerson = … (* Mitarbeiter oder Chef *);
case einePerson.Typfeld of
    Mitarbeiter: ihrEinkommen = einePerson.Gehalt;
    Chef: ihrEinkommen = einePerson.Firma.Einkommen
end
```

Bewertung der dynamischen Bindung:

- ☺ ungeheuer praktisch, da man sich viele case-Statements spart
- ☺ und damit schnell neue Unterklassen einführen kann
- ☹ gefährlich, da Aufrufer im Allgemeinen nicht weiss was für Code ausgeführt wird



Objekttypen:

- der **Typ** eines Objektes legt genau fest, welche Operationen auf dieses Objekt anwendbar sind
- Typen sind damit **Schnittstellendefinitionen** (Interfaces) für Objekte
- die Operationen (Methoden) einer Klasse bilden implizit einen Typ
- eine **statisch typisierte** Programmiersprache wie Java garantiert zur Übersetzungszeit, dass zur Laufzeit nie Methoden auf ein Objekt angewendet werden, dessen Klasse diese Methode nicht anbietet
- eine Klasse **implementiert** (besitzt) im allgemeinen mehrere Schnittstellen, ein Objekt besitzt damit im allgemeinen mehrere Typen
- Typen bilden damit **Sichten** auf Objekte (Klassen) für verschiedene Anwendungszwecke



Von Klassen zu Typen/Interfaces:



Klassen und Typen in Java realisiert:

Verkäuflich
(**Typ**)

gibPreis():Euro
…

implementiert

Abstraktion

Auto (**Klasse**)

preis : Euro
ort : String

gibPreis() : Euro
bewege(…)

implementiert

Abstraktion

Beweglich
(**Typ**)

bewege(…)
…

```
public class Auto implements Verkäuflich, Beweg-
lich  {
    private Euro preis;
    private String ort;
        …
```

```
public interface Verkäuf-
lich {
    public Euro gibPreis
```

```
public interface Beweg-
lich {
    public void bewege ( …
```



Mehrfachvererbung:

- eine Klasse kann in Java **mehrere „Interfaces“ implementieren**, sie erbt damit „quasi“ die zu implementierenden Operationen
- eine Klasse kann aber in Java nur **eine Oberklasse erweitern**, sie erbt
  - ⇨ Operationen mit Implementierungen
  - ⇨ Attribute mit Typdefinition
- in anderen OO-Programmiersprachen kann eine Klasse von mehreren Oberklassen erben, man spricht von „**Mehrfachvererbung**“
- dabei können **Vererbungskonflikte** auftreten:
  - ⇨ verschiedene Methoden mit dem selben Namen werden geerbt (Konfliktauflösung durch unterschiedliche Parametertypen)
  - ⇨ die selbe Methode mit unterschiedlich redefinierter Implementierung (Konflikte werden in OO-Programmiersprachen unterschiedlich aufgelöst)



Mehrfachvererbungskonflikt (durch Mehrfachdeklaration):



Mehrfachvererbungskonflikt (durch Redefinition):

Klasse	Attribute	Methode
Auto	…	String getFührerschein()
PKW (erbt von Auto)	…	String getFührerschein(){ return “Klasse3“; }
LKW (erbt von Auto)	…	String getFührerschein(){ return “Klasse2“; }
Van (erbt von PKW und LKW)	…	String getFührerschein(){ return ???; }



Varianten des Klassenbegriffs:

- **Klassen als Konzept**: sie werden zur Dokumentation von Begriffen/Konzepten eines Anwendungsbereiches (zusammen mit Glossar) verwendet
  - ⇨ in der Analysephase hauptsächlich verwendeter Klassenbegriff
- **Klassen als Objektmenge**: sie fassen eine Menge gleichartiger Objekte zusammen, ihre Extension; Extension einer Unterklasse ist Teilmenge der Extension ihrer Oberklasse
- **Klassen als Typ**: sie werden zur Festlegung der Schnittstelle (Operationen) und internen Struktur (Attribute) ihrer Objekte (Instanzen) eingesetzt; eine Unterklasse erweitert die Schnittstelle und Strukturdefinition ihrer Oberklasse
  - ⇨ vor allem in der Entwurfsphase verwendeter Klassenbegriff
- **Klassen als Implementierung:** eine Klasse stellt den Quellcode für ihre Objekte zur Verfügung; eine Unterklasse erbt den Quellcode ihrer Oberklasse und erweitert und überschreibt ihn soweit nötig (wurde hier in den Vordergrund gestellt)
  - ⇨ vor allem in der Implementierungsphase verwendeter Klassenbegriff



Kommunikation von Objekten:

- Objekte kommunizieren miteinander durch das Versenden von Nachrichten ("**message passing**")
- das sendende Objekt ("**sender**") verschickt eine Nachricht
- das empfangende Objekt ("**receiver**") erhält eine Nachricht
- der Empfänger besitzt eine Methode für die Nachrichtenbearbeitung

Zwei Arten von Nachrichten:

1. **Operationsaufruf**: Kommunikation zwischen einem sendenden und einem festgelegten empfangenden Objekt
   - ⇨ Empfänger führt Methode aus, die gerufene Operation implementiert
2. **Signal**: Kommunikation zwischen einem auslösenden Objekt und allen Objekten, die dieses Signal zum aktuellen Zeitpunkt empfangen können (oft über Kanal)
   - ⇨ Signal löst Aktion mit Zustandsänderung bei Empfänger aus



Zwei Arten der Kommunikation:

1. **synchrone Kommunikation:**
   der Sender wartet, bis der Empfänger die Nachricht verarbeitet hat (und einen Rückgabewert produziert hat)

2. **asynchrone Kommunikation:**
   der Sender wartet nicht, bis der Empfänger die Nachricht verarbeitet hat; benötigter Rückgabewert wird in separater Nachricht rückverschickt

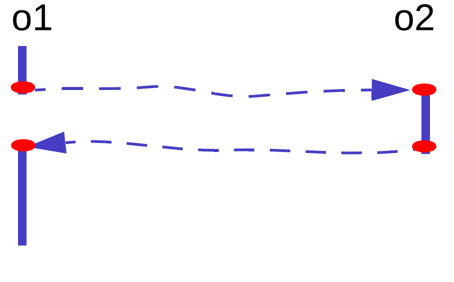
Kopplung Nachrichtenart ↔ Kommunikationsart:

- Operationsaufrufe sind sehr oft synchron, manchmal aber auch asynchron
- Signal-”Verschickung” ist fast immer asynchron, Synchronität wird höchstens dadurch erreicht, dass Sender auf zurückkommendes separates Signal wartet

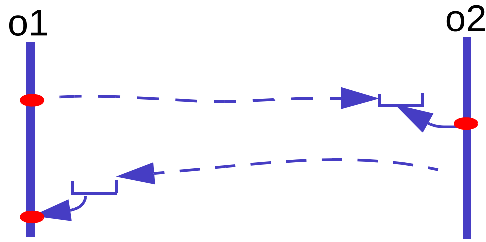


Zwei Ausführungsarten:

1. **sequentiell:**
   nur ein Objekt des Systems ist zu jedem beliebigen Zeitpunkt aktiv
   ("single thread of control")



2. **parallel:**
   mehrere Objekte sind zu einem bestimmten Zeitpunkt **gleichzeitig aktiv**
   ("multiple threads of control")



Beispiele für synchron/asynchron und sequentiell/parallel:

- Methodenaufruf in Java: synchron und sequentiell (Operationsaufruf)
- email: asynchron und parallel (Signalverschickung)
- telefonieren: synchron und parallel (Signalverschickung)



Parallelität und Nebenläufigkeit:

- von **parallelen Aktivitäten (parallel activities)** spricht man, wenn tatsächlich zwei verschiedene Objekte zeitgleich Operationen ausführen
- von **Nebenläufigkeit (concurrency)** spricht man, wenn zwei verschiedene Objekte möglicherweise parallel Operationen ausführen

Beispiel für Nebenläufigkeit:

Zwei verschiedene (aktive) Personen Timo und Hannah leihen sich ein Fahrrad (nebenläufig) aus; folgende Möglichkeiten der Abarbeitung gibt es:

1. Timo und Hannah leihen tatsächlich gleichzeitig ein Fahrrad aus (zwei Angestellte bedienen Timo und Hannah gleichzeitig)
2. Timo leiht vor Hannah sein Fahrrad aus (Hannah muss warten)
3. Hannah leiht vor Timo ihr Fahrrad aus (Timo muss warten)



Persistenz von Objekten:

Persistenz ist die Eigenschaft eines Objektes

- ⇨ die Lebenszeit des erzeugenden Programmes zu überleben (durch Serialisierung in Datei oder Speicherung in Datenbank)
- ⇨ von einem Adressbereich (Programm, Computer, …) zu einem anderen zu wechseln



4.3 Objektorientierte Programmier- und Modellierungssprachen:

Der “OO-Turm” von Babel:

Anfang der 90er Jahre gab es unzählige (etwa 50) miteinander konkurrierende OO-Modellierungsansätze samt der zugehörigen Diagrammarten und CASE-Umgebungen, die sich mal mehr, mal weniger voneinander unterschieden.

Beispiele:

❑ Object-Oriented Analysis (OOA):	Coad-Yourdon	1991
❑ Object Oriented Design (OOD):	**Booch**	1991
❑ Object Modeling Technique (OMT):	**Rumbaugh**	1991
❑ OO Software Engineering (OOSE):	**Jacobson**	1992
❑ Fusion:	Coleman	1994



Die Idee von Booch (Rational Software Corporation):

Vielzahl von OO-Modellierungsdialekten wird durch eine **Standard-Sprache** ersetzt, die **U**nified **M**odeling **L**anguage (**UML**), um so

- unnötige Grabenkriege in der OO-Gemeinde zu beerdigen
- die Vorteile verschiedener Modellierungsansätze zu vereinen
- potentiellen Anwendern die Qual der Wahl abzunehmen
- die Voraussetzung für Datenaustausch zwischen OO-Werkzeugen schaffen
- enge Integration von OO-Werkzeugen damit ermöglichen
- Abhängigkeit der Anwender von einem Hersteller zu reduzieren
- Werkzeugherstellern von der Unterstützung vieler Ansätzen zu befreien

Anmerkung:

Die Firma Rational wurde von IBM aufgekauft.



Die Historie des OO-Modellierungsstandards UML:

- Oktober 1994: Rumbaugh wechselt von GE zu Rational (Booch)
- Oktober 1995: Unified Method 0.8 = OMT (Rumbaugh) + OOD (Booch)
- Herbst 1995: Jacobson wechselt von Objective zu Rational — damit sind die "**Three amigos**" vereint
- Juni 1996: Unified Modeling Language 0.9 mit Use-Cases von Jacobson und Unterstützung durch DEC, HP, IBM, Microsoft, Oracle, …
- Januar 1997: UML 1.0 wird zur Standardisierung bei **OMG** = **O**bject **M**anagement **G**roup eingereicht
- September 1997: UML 1.1 mit erweiterter Beschreibung und Ergänzungen, insbesondere logikbasierte **O**bject **C**onstraint **L**anguage (**OCL**)
- November 1997: OMG akzeptiert überarbeitete UML als Standard
- August 2005: UML 2.0 wird als Standard verabschiedet
- seitdem:
  UML 2.1 (2007), UML 2.2 (2009), UML 2.3 (2010), UML 2.4 Beta (2011)



Stärken und Schwächen von UML:

- ☺ ein so genanntes Metamodell in Form von Klassendiagrammen legt relativ präzise die Syntax der unterstützten Diagrammarten fest
- 😐 die Semantik der Diagrammarten wird (noch) weitgehend in Prosa erläutert, einige Konsistenzregeln werden in OCL definiert
- ☹ es gibt etwa 12 Diagrammarten, die teilweise „gegeneinander" konkurrieren (verschiedene Diagramme können für ähnliche Zwecke eingesetzt werden)
- ☹ das Metamodell ist „leicht" erweiterbar, was uns eine Fülle von UML-Dialekten (Profile genannt) mit entsprechenden Werkzeugen beschert

Weiterführende Literatur zu UML:

Zu UML gibt es unzählige Bücher (sehr unterschiedlicher Qualität). Siehe Anfang des Vorlesungsskripts für Empfehlungen und Literaturliste am Ende dieses Kapitels!





Modell und Metamodell:

Typ (**Metaklasse**)

Klasse (**Metaklasse**)

Bestandteile meiner Modellierungssprache = Metamodell

Verkäuflich (**Typ**)
gibPreis():Euro
…

implementiert

Auto (**Klasse**)
preis : Euro
ort : String
gibPreis() : Euro
bewege(…)

implementiert

Beweglich (**Typ**)
bewege(…)
…

Modell meines Programms zur Entwicklungszeit

Instanz von

M - V 6120 : Auto
preis = 30.000
ort = “München“

DA - T 4378 : Auto
preis = 20.000
ort = “Darmstadt“

Objekte meines Programms zur Laufzeit



4.4 Weitere Literatur

[BHK04] M. Born, E. Holz, O. Kath: Softwareentwicklung mit UML2, Addison-Wesley (2004), 293 Seiten

Bereits Ende 2003 fertiggestelltes Buch über UML2, das deshalb die allerletzten Änderungen im Standard noch nicht nachvollzogen hat. Besonderheit: erläutert nicht nur umgangssprachlich UML2, sondern befasst sich auch mit der Definition der Sprache durch Metamodellierung (Modellierungselemente und Metamodellausschnitte werden abwechselnd präsentiert)

[Bu98] T. Budd: *Object-Oriented Programming with JAVA*, Addison-Wesley (1998)

Schöne Java-Einführung, die zudem die Konzepte der objektorientierten Programmierung erklärt.

[Bo94] G. Booch: *Object-Oriented Analysis and Design with Applications* (2nd Ed.), Benjamin/Cummings (1994)

Älteres Standardwerk zur objektorientierten Softwareentwicklung von dem „Haupt"-Vater der UML

[HK05] M. Hitz, G. Kappel, E. Kapsammer, W. Retschitzegger: *UML@Work*, 3-te Auflage, dpunkt.verlag (2005), 422 Seiten

Sehr schön geschriebenes UML-Buch mit durchgängigem Beispiel, das auch auf die „düsteren" Seiten der UML eingeht und umstrittene Sprachkonstrukte ausführlich diskutiert.



5. Objektorientierte Anforderungsanalyse

Themen dieses Kapitels:

- Beschreibung von Anwendungs-/Nutzfällen mit Use-Case-Diagrammen
- Datenmodellierung mit UML-Objekt- und Klassendiagrammen
- Modellierung von Abläufen mit UML-Aktivitätsdiagrammen
- Exkurs zu Petri-Netzen (semantische Basis von Aktivitätsdiagrammen)
- Vorschlag für den Aufbau eines Pflichtenheftes

Achtung:

- Ergebnis der Analysephase ist ein **Pflichtenheft** (erweitertes Lastenheft) mit UML-Diagrammen zur Beschreibung von Produktdaten und -funktionen
- Ausgangspunkt können ein (in der Machbarkeitsstudie erzeugtes) **Lastenheft** oder eine unstrukturierte Produktbeschreibung des Auftraggebers sein



5.1 Vom einfachen Lastenheft zum erweiterten UML-Pflichtenheft

Planung

Kundenanforderungen
(z.B. als Lastenheft)
(siehe Abschnitt 3.3)

Anwendungsfälle
→ UML Use Cases
(siehe Abschnitt 5.2)

"Workflow"-Modell
→ UML Activity Diagrams
(siehe Abschnitt 5.5)

Analyse

konzept. Datenmodell
→ UML Class Diagrams
(siehe Abschnitt 5.3)

feines Verhaltensmodell
→ UML Interaction Diag.
(siehe Abschnitt 7.3)

?

feines Klassenmodell
→ Class Diagram mit Ops.
(siehe Abschnitt 7.1)

?

reaktives Verhalten
→ UML State Machines
(siehe Abschnitt 7.4)

Analyse → Design



Einsatz von UML in der Analysephase:

- Ausgangspunkt sind die **Produktanforderungen** des Kunden (entweder als Fließtext oder als strukturiertes Lastenheft)
- Ergebnis ist ein (erweitertes) **Pflichtenheft** mit semiformalen Definitionen der Produktfunktionen und -daten als UML-Diagramme

Umfang des Einsatzes von UML umstritten:

- Extremposition eines **UML-Protagonisten**:
  - ⇨ UML wird als visuelle/grafische Programmiersprache eingesetzt
  - ⇨ erstelltes Analysemodell ist (als Prototyp) ausführbar
  - ⇨ man spricht vom ausführbaren Pflichtenheft
- Extremposition eines **UML-Antagonisten**:
  - ⇨ Pflichtenheft ist „nur“ in strukturiertem Englisch/Deutsch/… geschrieben
  - ⇨ grafische Modellierungssprachen wie UML werden erst im Design genutzt (wenn überhaupt - eigentlich reichen textuelle Programmiersprachen)



5.2 Produktfunktionen und Anwendungsfälle (UML Use Cases)

Aufgabe:

Ausgehend von den Produktfunktionen im Lastenheft oder einer Fließtext-beschreibung des Softwareprodukts werden seine Funktionen anhand konkreter Fallbeispiele = **Szenarien** detailierter beschrieben. Verwandte Szenarien werden zu Produktfunktionen = **Anwendungsfälle** zusammengefasst (siehe auch Techniken zur Ermittlung von Anforderungen in Abschnitt 3.2).

Vorgehensweise:

- Skizze konkreter Benutzungsszenarien mit Akteuren (mit Rollenspielen)
- Identifikation der Grenze zwischen Softwaresystem und „Außenwelt”
- Skizze der zugehörigen Benutzeroberfläche (ggf. Prototypbau)
- Zusammenfassung ähnlicher Szenarien zu Anwendungsfällen (Use Cases, manchmal auch Nutzfälle genannt)
- Überblick über Anwendungsfälle durch Anwendungsfalldiagramme



Bestimmung geeigneter Szenarien oder Anwendungsfälle:

- man nehme die Produktfunktionen aus dem Lastenheft (verschiebt Problem)
- alle **Verben** in einer Softwareproduktbeschreibung markieren:

Motor Vehicle Reservation System

A rental office lends motor vehicles of different types. The assortment comprises cars, vans, and trucks. Vans are small trucks, which may be used with the same driving license as cars. Some client may reserve motor vehicles of a certain category for a certain period. He or she has to sign a reservation contract. The rental office guarantees that a motor vehicle of the desired category will be available for the requested period. The client may cancel the reservation at any time. When the client fetches the motor vehicle he or she has to sign a rental contract and optionally an associated insurance contract. Within the reserved period, at latest at its end, the client returns the motor vehicle and pays the bill.



Bestimmung geeigneter Anwendungsfälle - 2:

- auf relevante Verben einschränken:



Systemgrenze und Anwendungsfälle des MVRS-Beispiels:



Oberflächenbau mit GUI-Werkzeug:

Together 6 -- MVRS

Client Number: 100000000000
Reservation Date: Day/Month/Year
Car Category:
ok cancel

Bestandteile der UI-Klasse

erzeugter Button

interaktives Oberflächendesign

GUI-Designelemente

Eigenschaften von „Button“

Name	Value
actionCommand	ok
background	
font	Dialog 12
foreground	
label	ok
size	Width: 57 Height: 25

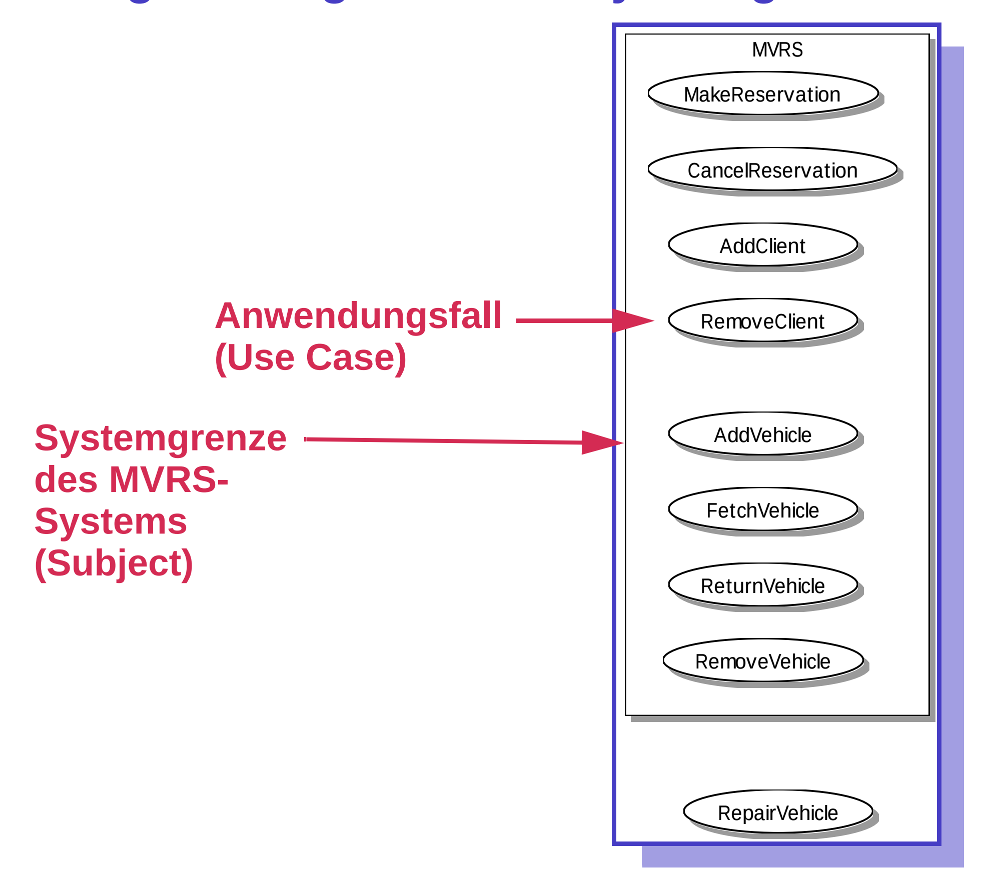
```
        System.exit(0);
    }

    private Label label1 = new Label();
    private TextField textField1 = new TextField();
    private Label label2 = new Label();
    private TextField textField2 = new TextField();
    private Label label3 = new Label();
    private Choice choice1 = new Choice();
    private Button button1 = new Button();
    private Button button2 = new Button();
}
```

generierter Code



Anwendungsfalldiagramm mit Systemgrenze:



RepairVehicle ist eine Funktion, die vom MVRS-System in keiner Weise unterstützt wird (im geplanten Release), und deshalb außerhalb des MVRS-Kastens.



Anwendungsfalldiagramm mit Akteuren:

Akteur
(Actor)

„benutzt"
(Association)

MVRS

MakeReservation

CancelReservation

AddClient

RemoveClient

AddVehicle

FetchVehicle

ReturnVehicle

RemoveVehicle

RepairVehicle

Clerk

Owner

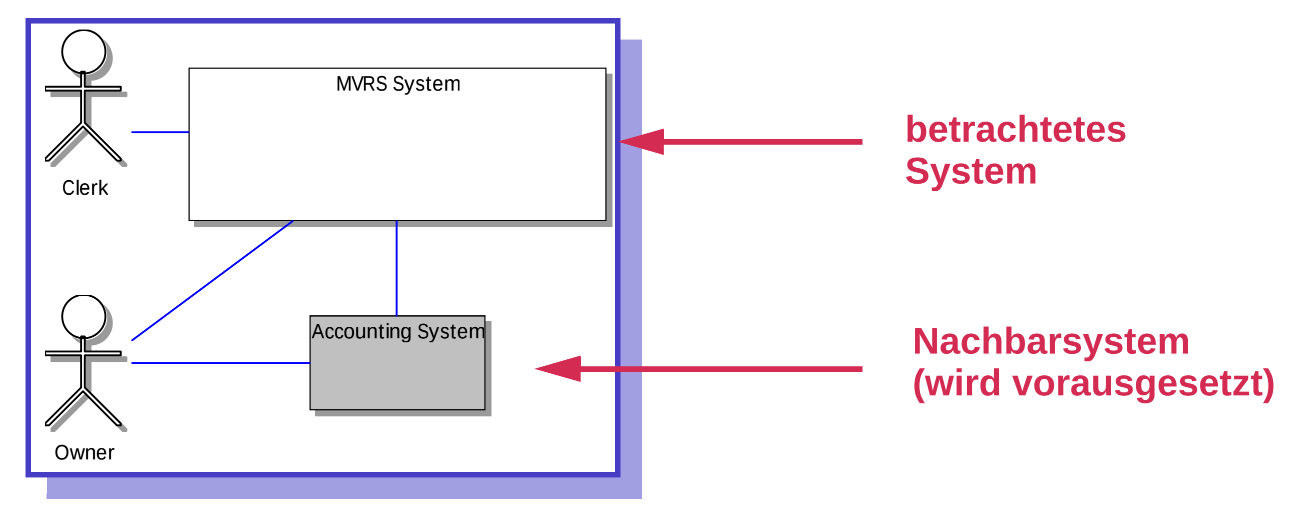


Kommentare zu „Benutzt“-Beziehungen (Assoziationen):

- ☺ Akteure sind die spätere Benutzer des Systems (in verschiedenen Rollen)
- ☺ die Linie von Akteur (Actor) zu Anwendungsfall (Use Case) erlaubt Aufruf eines Anwendungsfalles durch einen Akteur (Benutzung der Systemfunktion)
- ☹ interagieren mehrere Akteure mit einem Anwendungsfall, so ist nicht festgelegt, was das bedeutet (wie bei MakeReservation):
  - ⇨ meist (hier): jeder Akteur für sich darf die Systemfunktion benutzen
  - ⇨ ggf. aber auch: nur alle Akteure dürfen die Systemfunktion nutzen
- ☹ unklar ist zunächst auch, wieviele Funktionen (Anwendungsfälle) ein Anwender gleichzeitig benutzen darf (meist: keine Einschränkungen vorgesehen)
  - ⇨ sind gleichzeitiges Make- und CancelReservation erlaubt
  - ⇨ dürfen mehrere Reservierungen „gleichzeitig“ durchgeführt werden
- ☹ die vielen Beziehungen im vorigen Beispiel machen das Diagramm unübersichtlich



Systemkontextdiagramm als Variante des Anwendungsfalldiagramms:



- Kontextdiagramm beschreibt die Umgebung des zu entwickelnden Systems
- es gibt verschiedene Möglichkeiten in UML, Kontextdiagramme zu „malen“
- in der Analysephase ist es am sinnvollsten, sie als Spezialfall von Anwendungsfalldiagrammen zu aufzufassen
- Accounting System ist ein System, das mit „unserem“ MVRS System interagiert und bereits existiert (oder später realisiert wird)

Fachgebiet Echtzeit-systeme



Zerlegung in Teilsysteme zur Strukturierung von Anwendungsfällen:

MVRS_Clients

AddClient

RemoveClient

Clerk

MVRS_Reservations

MakeReservation

CancelReservation

MVRS_Vehicles

AddVehicle

FetchVehicle

ReturnVehicle

RemoveVehicle

Owner

Erstes Release hat noch keine separate Verwaltung von Kundenstammdaten

Kommentare

Fahrzeug wird für einen bestimmten Zeitraum reserviert

Reservierung wird gelöscht

...

Beziehungen ausgeblendet

Neues Fahrzeug wird in Fuhrpark aufgenommen

Fahrzeug wird abgeholt

Fahrzeug wird zurückgebracht

Fahrzeug wird ausgemustert



Kommentare zu der Zerlegung in Teilsysteme:

- man muss sich gut überlegen, ob die Zerlegung eines Systems in Teilsysteme in der Analysephase bereits sinnvoll ist
  - ⇨ eigentlich geht es ja um das „was bietet das System an" und nicht um das „wie wird das System realisiert"
- **aber**: für die Projektplanung ist eine frühe grobe Einteilung eines Systems in Teilsysteme und Zuordnung von Funktionen zu Teilsystemen sinnvoll
- zudem ist es eine naheliegende Möglichkeit, eine große Menge von Anwendungsfällen sinnvoll zu unterteilen
  - ⇨ eine andere Variante der Unterteilung (Hierarchisierung) werden wir am Ende des Abschnitts kennenlernen (mit Paketen)



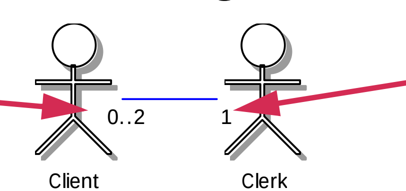
Externe Beziehungen zwischen und Generalisierung von Akteuren:



Kommentare zu Beziehungen zwischen Akteuren:

- solche Beziehungen sieht man nur selten, da in der Regel nur die Akteure modelliert werden, die mit dem System **direkt** interagieren
- manchmal ist es aber sinnvoll, auch indirekte „Kunden“ eines Systems aufzuführen und sie mit den eigentlichen Akteuren zu verbinden
- die Beziehung (Assoziation) zwischen Client und Clerk deutet an, dass Clerks bei der MVRS-Systembenutzung mit Clients interagieren

wieviele Clients inter-agieren mit einem Clerk gleichzeitig



wieviele Clerks inter-agieren mit einem Client gleichzeitig

- die zusätzliche Multiplizitäten (Zahlen an der Beziehungslinie) deuten an, dass
  - ⇨ ein Clerk mit maximal 2 Clients (gleichzeitig) interagieren darf aber auch mal keinen Kunden haben kann
  - ⇨ ein Client mit genau einem Clerk interagiert (das ist der Standard-fall der normalerweise nicht dargestellt wird)



Kommentare zur Generalisierungsbeziehung:

- ein Owner ist ein spezieller Clerk, der alle Eigenschaften eines Clerk besitzt (erbt) und zusätzliche Eigenschaften haben kann
- ein speziellerer Akteur darf alle die Systemfunktionen benutzen (Anwendungsfälle), die der allgemeinere Akteur benutzen darf
- ein spezieller Akteur kann mit allen Akteuren interagieren, mit denen der allgemeinere Akteur interagiert
- die Generalisierung zwischen Akteuren ist ein Spezialfall der Generalisierung zwischen „klassenartigen“ Modellierungselementen in UML (als Classifier bezeichnet)
- die Vererbung auf Java-Klassen (extend) ist ein Spezialfall der Generalisierung von UML



Generalisierungsbeziehungen zwischen Anwendungsfällen:

MVRS_Clients
ManipulateClient
AddClient
RemoveClient

abstrakter Anwendungsfall
Generalisierung

0..2
1
Client
Clerk
Owner

MVRS_Reservations
ManipulateReservation
MakeReservation
CancelReservation

MVRS_Vehicles
ManipulateVehicle
AddVehicle
FetchVehicle
ReturnVehicle
RemoveVehicle

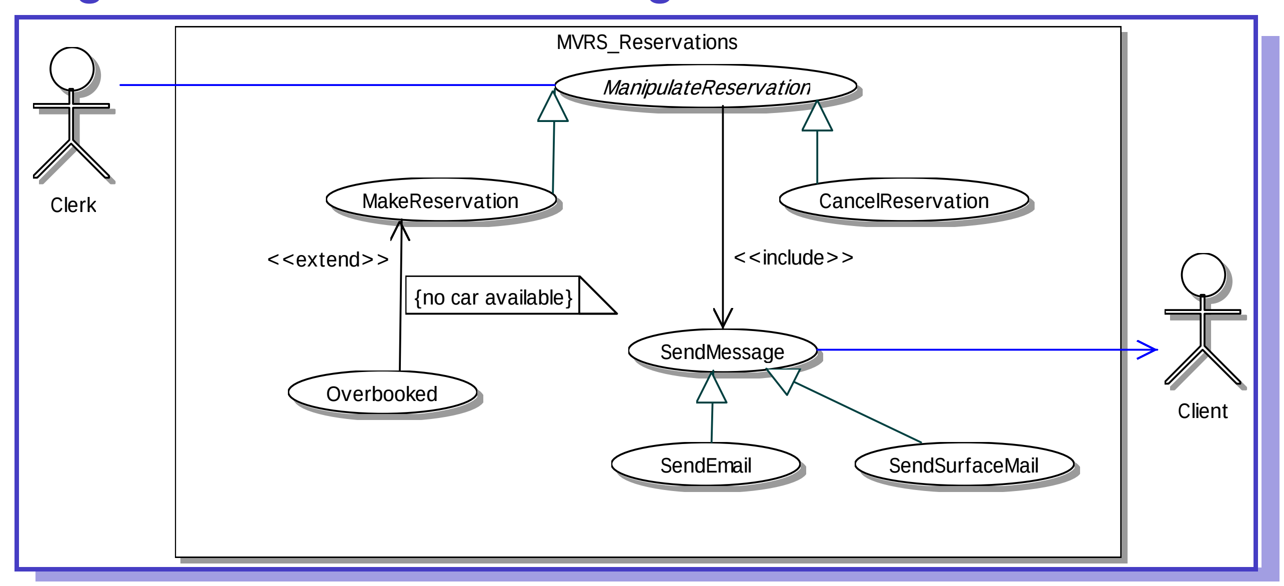


Kommentare zu Generalisierung von Anwendungsfällen

- Anwendungsfälle mit gemeinsamen Eigenschaften (wie Beziehungen zu Akteuren) können von einem generalisierten Anwendungsfall abgeleitet werden (und die Eigenschaften von diesem erben)
- man kann so Diagramme übersichtlicher gestalten, sinnvolle Gruppierungen von Anwendungsfällen schaffen und sich Schreibarbeit sparen
- zudem kann man so beispielsweise zum Ausdruck bringen, dass Clerk zu jedem Zeitpunkt nur eine der beiden Systemfunktionen AddClient oder RemoveClient verwenden kann
- generelle Anwendungsfälle wie ManipulateClient können **abstrakt** sein (kursiver Name), wenn sie keine eigenständigen Systemfunktionen darstellen (sondern nur zur Vererbung von Eigenschaften eingeführt wurden)



Weitere Beziehungen zwischen Anwendungsfällen:



- **extend:** Varianten (Ausnahmen) eines Anwendungsfalles werden als eigene Fälle beschrieben; es wird also Funktionalität zu einem Anwendungsfall hinzugefügt; der erweiternde Fall kennt dabei den Basisanwendungsfall, aber nicht umgekehrt
- **include:** besitzen mehrere Anwendungsfälle gleiche Teilabläufe wie das Nachrichtenverschicken, so können diese Teilabläufe separat beschrieben werden; es wird also Funktionalität ausgelagert und wiederverwendet bzw. aufgerufen; der Aufrufer kennt den aufgerufenen Anwendungsfall



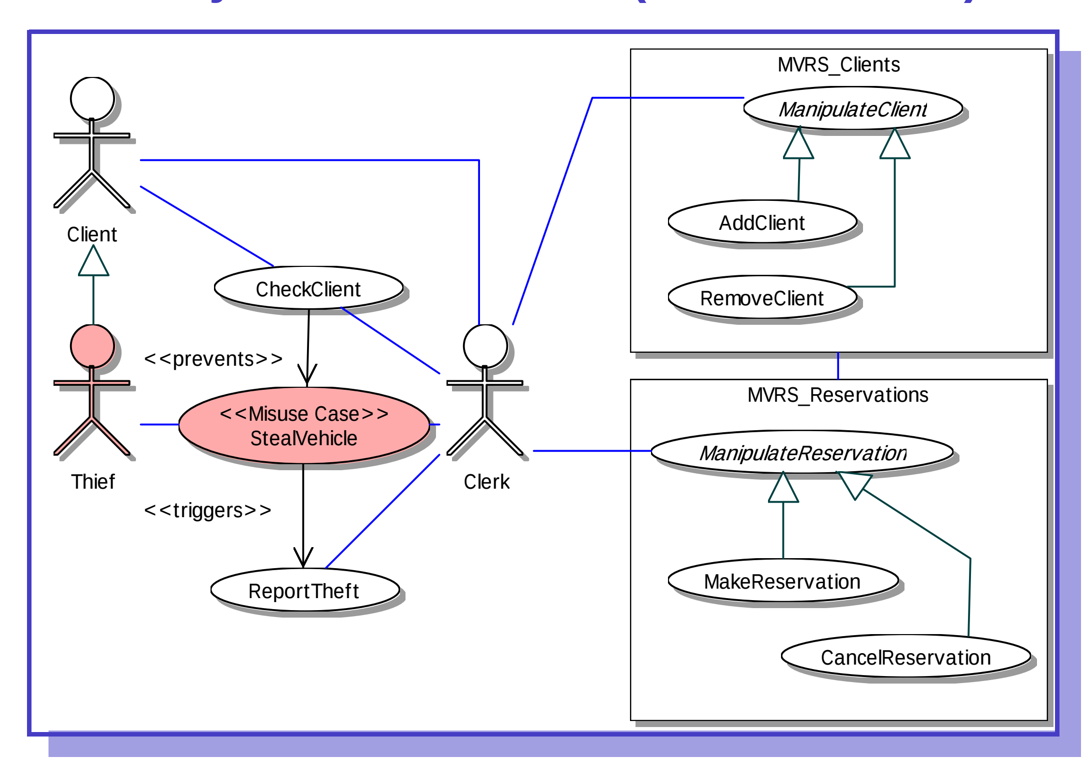
Exkurs zu Diagrammannotationen der Form ‹‹xxx›› :

Annotationen der Form ‹‹include›› werden „**Stereotypen**" genannt und erweitern die Sprache UML (präziser: das **Metamodell** von UML):

- sie erlauben die Unterscheidung verschiedener Unterarten eines Modellierungselementes (z.B. verschiedene „Benutzt"-Beziehungen zwischen Anwendungsfällen)
- manche Stereotypen sind durch den UML-Standard vorgegeben
- es gibt Standarderweiterungen für bestimmte Anwendungsbereiche (z.B. Modellierung von Echtzeitsystemen) mit weiteren Stereotypen
- schließlich kann der UML-Anwender selbst neue Stereotypen einführen (und ggf. eine neue Darstellung definieren)
- Definition neuer Stereotypen wird von UML-Werkzeugen sehr unterschiedlich unterstützt



Anwendungsfälle für Systemmissbrauch (Misuse Cases):



- ☹ die Funktion CheckClient soll den Autodiebstahl im Vorfeld verhindern
- ☹ die Funktion ReportTheft wird benötigt, falls doch …
- ☹ die Stereotypen Misuse Case, prevents und triggers sind „Eigenerfindungen“



Detaillierte textuelle Anwendungsfallbeschreibung:

Use Case: MakeReservation

Actors: Clerk (plus ggf. beteiligte Teilsysteme)

Purpose: Fahrzeug der Kategorie C wird für Zeitraum T reserviert.

Entry Cond: Reservierungsfunktion wird aktiviert (geht immer)

Overview: Aufgrund des Telefonanrufes, der schriftlichen Bestellung oder des persönlichen Erscheinens eines Kunden K (Client) gibt Clerk eine Reservierung mit Fahrzeugkategorie C und Zeitraum T ein. Diese Reservierung wird - soweit möglich - berücksichtigt und es wird eine Bestätigung dem Kunden zugeschickt.

Exit Cond: gewählte Reservierung in DB oder unveränderte DB

Includes: SendMail (für Verschicken einer Reservierungsbestätigung)

Special Req: Die Suche nach verfügbaren Fahrzeugen dauert nicht länger als 30 Sekunden.

Category: sehr hohe Priorität

Cross Ref: auf /LF 30/ aus Lastenheft (siehe Abschnitt 3.3)



Textuelle Ablaufbeschreibung:

Actor Action	System Response
1. Anwendungsfall beginnt mit Aufruf der Hauptfunktion (1) im Menü	
2. gibt die Nummer eines Kunden ein (wird ggf. vorab bestimmt)	3. sucht Client in Datenbank und erfragt Reservierungszeitraum T
4. gibt Zeitraum T ein	5. erfragt Fahrzeugkategorie K
6. gibt Fahrzeugkategorie K ein	7. bestimmt freie Fahrzeuge zu T und K
8. wählt aus freien Fahrzeugen eines (nach Kundenwunsch) aus	9. trägt Reservierung in Datenbank ein und ruft Anwendungsfall SendMail

Achtung:

Fallunterscheidungen, Iterationen und andere Programmierkonstrukte vermeiden.
Lieber Varianten eines Ablaufs als eigene Anwendungsfälle beschreiben.



Erweiterung eines Anwendungsfalles:

Use Case: Overbooked

Purpose: Ausnahmebehandlung falls Reservierungswunsch nicht erfüllbar.

Extends: ReserveVehicle

At Point: Nach Schritt 7 von ReserveVehicle

Entry Cond: Schritt 7 hat leere Liste von Fahrzeugen geliefert

Overview: Es wird für den vorgegebenen Zeitraum T eine Liste der Fahrzeuge der nächsthöheren Kategorie K + 1 bestimmt.

Exit Cond: potientiell wieder leere Liste von Fahrzeugen wurde berechnet

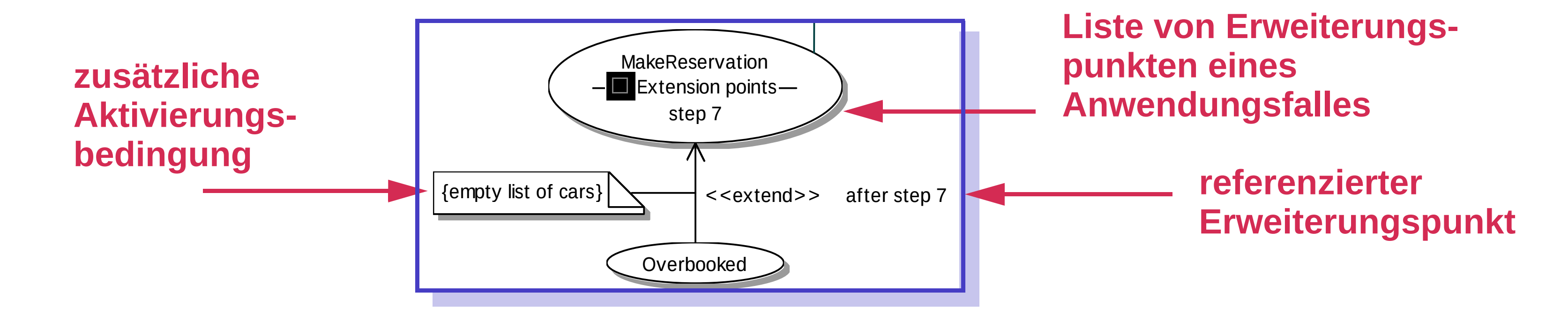
Return To: Aufrufstelle in ReserveVehicle, falls Liste nicht leer
Ende von ReserveVehicle, falls Liste leer

Category: niedrige Priorität (erspart nur Neustarten der Reservierung)

Cross Ref: auf /LF 30/ aus Lastenheft (siehe Abschnitt 3.3)



Notation von Erweiterungspunkten:



Problem mit Erweiterungspunkten:

- der erweiterte Anwendungsfall soll nichts von seinen Erweiterungen „wissen“
- also muss er an jeder beliebigen Stelle erweiterbar sein (extension points besitzen)
- führt zu einer nichtssagenden langen Liste von Erweiterungspunkten
- Syntax für die Referenzierung von Erweiterungspunkten nicht im Standard (after, before, ...)
- viele Werkzeuge unterstützen das alles nur mit „Trickserei“



Variante der Anwendungsfallbeschreibung aus [St05]:

Es gibt nicht die Standardstruktur für die textuelle Beschreibung von Anwendungsfällen, sondern jedes Buch (jede Firma) hat ihr eigenes „Template“ dafür. So gibt es in [St05] beispielsweise folgende Zusatzfelder:

- statt entry condition wird feiner zwischen **Auslöser** eines Anwendungsfalles und **Vorbedingung** für die Ausführbarkeit des Anwendungsfalles unterschieden
- statt exit condition wird feiner zwischen **Nachbedingung** eines Anwendungsfalles (was gilt danach immer) und **Ergebnis** des Anwendungsfalles unterschieden (was hat der Anwendungsfall bewirkt)
- anstatt einer einzigen Ablaufbeschreibung je Anwendungsfall werden oft mehrere Ablaufbeschreibungen (Standardablauf und Ausnahmen/Varianten) in einem Anwendungsfall zusammengefasst und damit auf extend-Beziehung verzichtet
- sinnvoll können des weiteren Felder für Häufigkeit des Auftretens eines Anwendungsfalles sowie allgemeine Anmerkungsfelder sein
- auch für textuelle Darstellung von Abläufen gibt es die verschiedensten Varianten (z.B. tabellarische Form mit beteiligten Akteuren je Teilschritt, ...)



Kategorisierung (ranking) von Anwendungsfällen:

Die Vergabe von **Prioritäten** an Anwendungsfälle wird für die **Projektplanung** benutzt. Anwendungsfälle mit höherer Priorität werden früher realisiert als Anwendungsfälle mit niedrigerer Priorität.

Anwendungsfälle haben hohe Priorität, wenn

- sie großen Einfluss auf Architektur des Softwareproduktes besitzen
- zeitkritische, sicherheitskritische, komplexe, … Funktionen beschreiben
- mit Hilfe neuer (riskanter) Technologien zu realisieren sind
- (über-)lebensnotwendige Geschäftsfunktionen darstellen
- sie erhöhten Gewinn versprechen
- [einfach zu realisieren sind]



Anmerkungen zu Anwendungsfällen:

- **wiederkehrende Teile** von Anwendungsfällen werden über include-Beziehung in eigene Anwendungsfälle ausgelagert (aber nur wenn wirklich notwendig)
- die **include-Beziehung** hat die Semantik eines Prozeduraufrufes, der Aufruf wird in der Beschreibung des rufenden Anwendungsfalles niedergeschrieben
- ein Anwendungsfall beschreibt den Normalfall eines konkreten Ablaufes, **Ausnahmefälle** oder Varianten werden über extends-Beziehung ausgelagert
- ein Anwendungsfall abstrahiert von einer Anzahl konkreter **Szenarien** (wie etwa „Kunde Franklin Turtle reserviert für den 28. April 2006 einen Smart")
- statt textueller Beschreibung eines Ablaufs werden auch **Sequenzdiagramme** (siehe Abschnitt 7.3) oder **Aktivitätsdiagramme** (siehe Abschnitt 5.5) eingesetzt
- die **extends-Beziehung** entspricht einem bedingten, versteckten Prozeduraufruf: der erweiternde Anwendungsfall wird an einem Punkt (oder in einem Bereich) des skizzierten Ablaufs des Basisanwendungsfalles aktiv, wenn Bedingung erfüllt ist
- jeder Anwendungsfall sollte auf eine **Lastenheftfunktion** Bezug nehmen und in seiner Beschreibung Begriffe aus **Glossar** verwenden



Probleme bei der Verwendung von Anforderungsfällen:

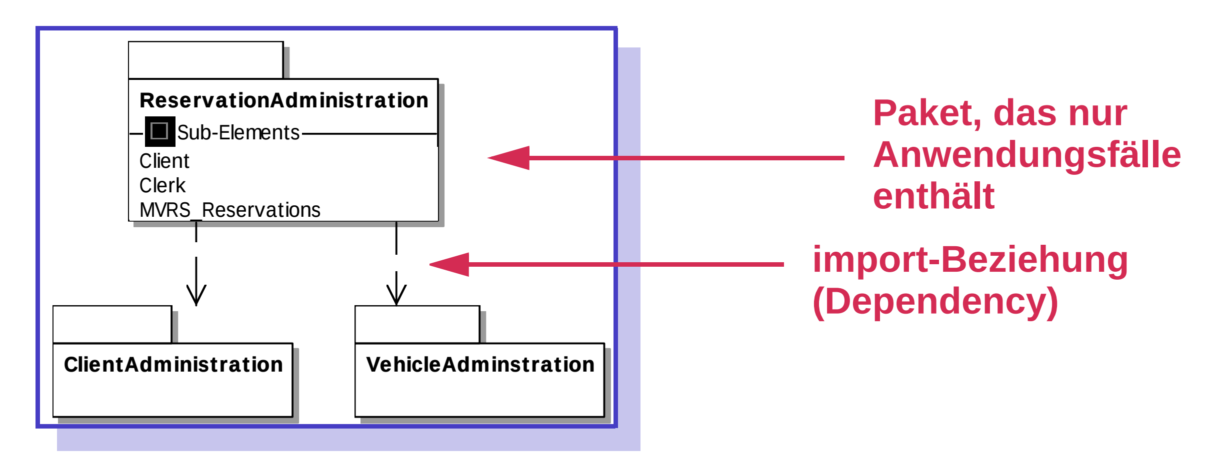
- Wie findet man Anwendungsfälle und wer sind die Akteure?

Achtung:

Manche Experten raten zur Verwendung weniger Anwendungsfälle und Vermeidung von



Pakete zur hierarchischen Strukturierung von Anwendungsfällen:



Paket, das nur Anwendungsfälle enthält

import-Beziehung (Dependency)

- **Paketdiagramme** wurden eigentlich für die Organisation vieler Klassen eines Projektes eingeführt (wie „packages“ in Java)
- sie werden aber inzwischen für die **hierarchische Organisation** beliebiger UML-Artefakte eingesetzt
- mit ihrer Hilfe lassen sich also auch die Anwendungsfälle eines Projekts übersichtlich anordnen
- weiteres hierzu in Kapitel 8



Zusammenfassung der Anwendungsfalldiagrammelemente:

(Sub)Systemgrenze oder Partition

<<primary>>
Anwendungsfall auslösender Akteur

Generalisierung auf Akteuren

<<secondary>>
für Anwendungsfall zusätzlicher Akteur

löst aus und nutzt

ebenfalls beteiligt

<<primary>>
Anwendungsfall
Erweiterungspunkt für Ausnahme

<<include>>

<<secondary>>
Hilfsanwendungsfall

ruft auf

Generalisierung auf Anwendungsfällen

Spezialisierung

<<extend>>

{Aktivierungsbedingung}

<<secondary>>
Ausnahmeanwendungsfall

Es handelt sich um eine ziemlich vollständige Aufzählung aller UML-Sprachelemente, die in Anwendungsfalldiagrammen benutzt werden können. Die Unterscheidung von primären und sekundären Akteuren und Anwendungsfällen wird dabei eher selten verwendet.



5.3 Produktdatenmodellierung (UML Object/Class Diagrams)

Aufgabe:

Ausgehend von den Produktdaten im Lastenheft oder einer Fließtextbeschreibung des Softwareprodukts und den Anwendungsfällen wird ein **konzeptuelles Datenmodell** (**Domänenmodell**) des Anwendungsbereiches erstellt. Es handelt sich dabei nicht um das interne Datenmodell des später realisierten Softwareproduktes!

Vorgehensweise:

- ⇨ Bestimmung von Objekt- bzw. Klassenkandidaten mit verschiedenen Methoden (Unterstreichungen im Text, Akteure, Glossar)
- ⇨ Bestimmung wichtiger Assoziationen (Beziehungen) und Attribute zu den identifizierten Klassen (optional: Vererbungsbeziehungen)
- ⇨ ggf. zunächst Erstellung konkreter(er) Objektdiagramme und dann Erstellung abstrakter(er) (Analyse-)Klassendiagramme
- ⇨ meist werden noch keine Operationen bei Klassen festgelegt
- ⇨ alle gefundenen Klassen (Beziehungen?) auch im Glossar aufführen!



Von der realen Welt zum Klassendiagramm:

Klassendiagramm

Client
Attributes
name
frequentClient

has

Contract
Attributes
contractNo

Attribute

Klasse

worksFor

Assoziation

Company
Attributes
name

Abstraktion

Objektdiagramm

Helga:Client
name = "Helga"

has

:Contract

Link

worksFor

DeadFishInc

Objekt

Arne:Client
name = "Arne"

worksFor

has

789:Contract

reale Welt

Abstraktion

Dead Fish Inc.

Arne

Helga

...

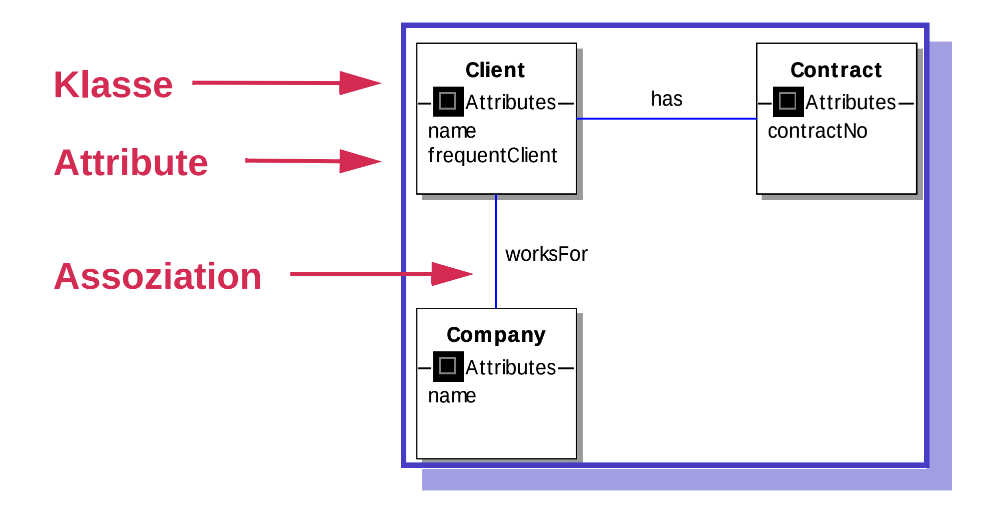
007

123

789



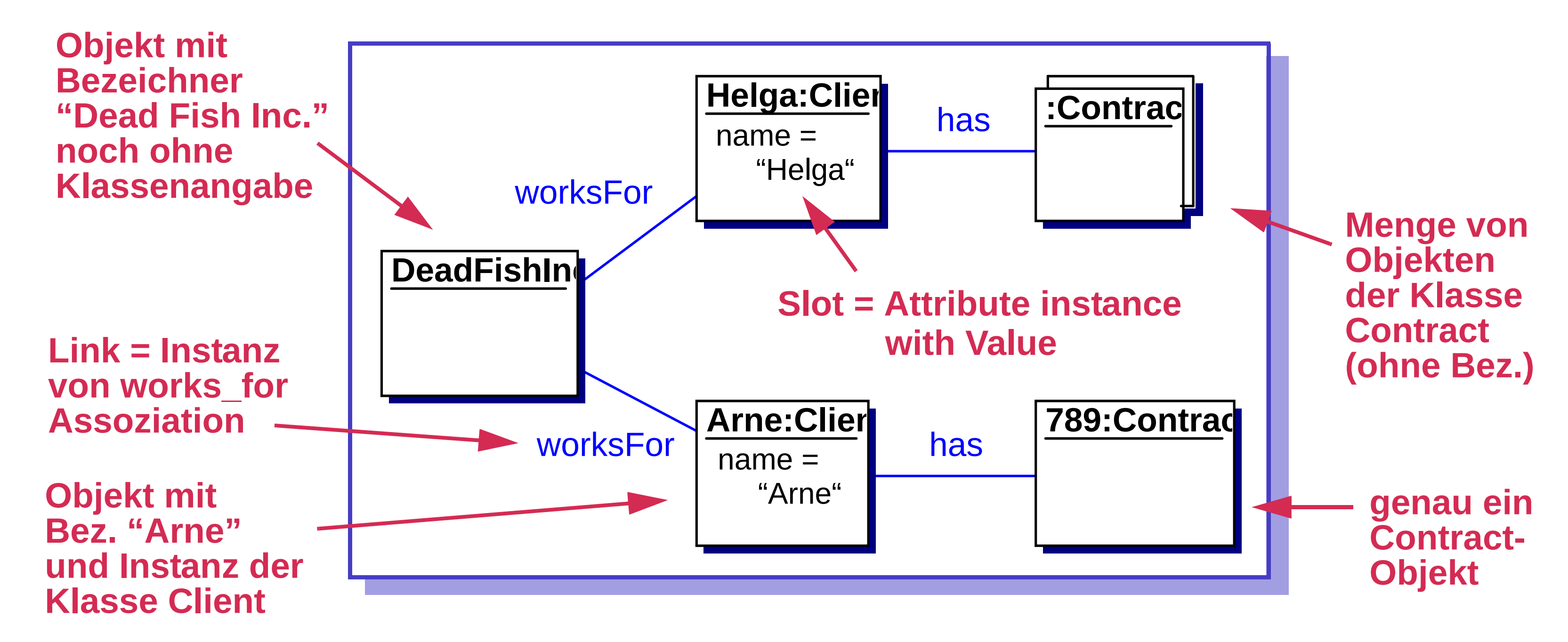
Einfaches Klassendiagramm:



- Ursprung: Entity-Relationship-Modell (siehe Abschnitt 4.1, Seite 108)
- es legt die Objekt- und Beziehungsarten (Assoziationen) des Anwendungsbereichs mit ihren wichtigsten Attributen fest
- Klassen werden in dieser Phase (meist) ohne Operationen definiert
  - ⇨ Grund:



Einfaches Objektdiagramm



- Modellierung von konkreten Szenarien (Snapshots) mit Klasseninstanzen (Instance Specifications) = **Objekten** und Assoziationsinstanzen = **Links**
- Gegenstück der Datenmodellierung zu den funktionalen Anwendungsfällen
- manchmal hilfreich für den Übergang von der realen Welt zu den abstrakten Klassendiagrammen



Kandidaten für Objekte und Klassen:

- alle phyischen, berühbaren Gegenstände (wie Vehicle)
- Personenrollen (wie Clerk, Client)
- Institutionen (wie Company)
- abstrakte Begriffe (wie Address)
- durchzuführende Transaktionen (wie Renting, Payment)
- eintretende Ereignisse (wie Accident)
- …

Man beachte:

Auch Operationen, Transaktionen, Ereignisse, Prozesse, … können sinnvolle Objektkandidaten sein. Nicht nur passive Datenbestände werden als Objekte modelliert!



Identifikation von Klassenkandidaten im Fließtext:

- alle **Substantive** in einer Softwareproduktbeschreibung markieren:

Motor Vehicle Reservation System

A rental office lends motor vehicles of different types. The assort-ment comprises cars, vans, and trucks. Vans are small trucks, which may be used with the same driving license as cars. Some client may reserve motor vehicles of a certain category for a certain period. He or she has to sign a reservation contract. The rental office guaran-tees that a motor vehicle of the desired category will be available for the requested period. The client may cancel the reservation at any time. When the client fetches the motor vehicle he or she has to sign a rental contract and optionally an associated insurance contract. Within the reserved period, at latest at its end, the client returns the motor vehicle and pays the bill.

? Synonyme identifizieren (wie Type und Category in dem obigen Text)



Identifikation von Klassenkandidaten im Fließtext - 2:

- irrelevante Klassenkandidaten streichen:



Aufbau von Klassen:

Name → ReservationContract ← falls Name kursiv wäre, dann **abstrakte** Klasse

Attribute (ggf. noch ohne Attributtypen) →
contractId
category
deposit
period

Operationen → **werden in Kapitel 7 eingeführt**

Abstrakte Klassen sind Klassen, die keine eigenen Objekte besitzen, sondern nur ihre Eigenschaften an Unterklassen vererben können. Nur abstrakte Klassen können abstrakte Operationen (ohne Implementationen) besitzen (siehe Abschnitt 7.1)



Identifikation von Attributen:

- Attribute “speichern” **Eigenschaften** von Objekten als **Werte** (und normalerweise nicht als eigenständige Objekte)
- ihre **Attributwerte** sind Elemente von Attributdatentypen
- sinnvolle **Attributdatentypen** sind
  - ⇨ primitiver Datentypen wie Integer und String oder
  - ⇨ Aufzählungstypen wie Color (Black, White, …) oder
  - ⇨ einfache zusammengesetzte Typen wie Address und Date (Achtung: Datum und Adresse sind aus Modellierungssicht keine Objekte, da Adressen oder Datumsangaben mit unterschiedlichen Werten auch verschiedenen sind; sie haben keine eigene Objektidentität)
- Attribute sollten bei der Modellierung **nie Zeiger** auf andere „wichtige“ Objekte sein, also keine Klassen als Typ besitzen (dafür gibt es Assoziationen)



Beispiele für Attributdeklarationen:

Client
Attributes
-clientId:Number
-surname:String
-firstNames:StringSet
-Birthday:Date

ReservationContract
Attributes
-contractId:Number
-period:DateRange
-category:CatType
-deposit:Currency

Vehicle
Attributes
-vehicleId:Number
-category:CatType

Sichtbarkeit „-“ = „private” (außen nicht sichtbar)

Multiplizitätsangabe: oft auch als „firstNames: String [1..3]
Menge von Strings (mindestens einer, höchstens drei)

Achtung: Sichtbarkeiten werden hier zwar kurz eingeführt, aber erst im folgenden Kapitel 8 genauer diskutiert (gehören zum Design und nicht zur Analyse).



Aufbau einer Attributdeklaration (UML-Standard):

<Sichtbarkeit> <Name> : <Typ> <Multiplizität> = <Wert> {<Eigenschaften>}

- **Sichtbarkeit**: definiert Sichtbarkeit eines Attributs außerhalb der Klasse
  - ⇨ **public** oder „**+**": expliziter schreibender und lesender Zugriff von außen erlaubt; das sollte man eigentlich nie verwenden
  - ⇨ **protected** oder „**#**": nur Klasse selbst und ihre Unterklassen dürfen in ihren Methoden auf das Attribut zugreifen
  - ⇨ **private** oder „**-**": nur Implementierung der Klasse selbst darf auf Attribut schreibend und lesend zugreifen
  - ⇨ **package local** oder „**~**": alle Klassen innerhalb desselben Pakets können auf das Attribut zugreifen.
- **Name**: irgendeine Attributbezeichnung, die innerhalb der Klasse eindeutig ist
  - ⇨ ist dem Namen ein „**/**" vorangestellt, so ist es ein **abgeleitetes** Attribut (Wert bestimmt sich durch Rechenvorschrift und nicht durch Zuweisung)



Aufbau einer Attributdeklaration - 2:

- **Typ** eines Attributs:
  - in Programmiersprache definierter Typ wie int, float, …
  - Name einer „Hilfs“-Klasse für Attributwerte, die (einfache) Objekte sind
- **Multiplizität** mit Aufbau [<untere Grenze> .. <obere Grenze>] oder [<Zahl>]:
  - [0..1]: optionales Attribut mit erlaubtem “null”-Wert
  - [0..*]: mengenwertiges Attribut (auch leere Menge)
  - [m..n]: Attributmenge mit m bis n Elementen
  - [n]: genau n Elemente enthält das mengenwertige Attribut
  - [1]: genau ein Wert, der Normalfall (wenn keine Multiplizität sichtbar ist)
- **Wert** = Rechenvorschrift für Attribut, oft eine Konstante
  - bei normalen Attributen gibt Ausdruck Initialwert an
  - bei abgeleiteten Attributen die Berechnungsvorschrift



Aufbau einer Attributdeklaration - 3:

- Menge von **Attributeigenschaften** wie etwa:
  - ⇨ **readonly**: auf Attribut nur lesend nach Initialisierung zugreifen
  - ⇨ **unique**: das Attribut ist eine Kollektion, deren Elemente alle nur einmal auftreten
  - ⇨ **ordered**: das Attribut ist eine Kollektion, deren Elemente in einer bestimmten Reihenfolge gespeichert sind

Kombinationen von unique und ordered:

- unique = true und ordered = true: es handelt sich um eine Liste von Elementen, die jeweils nur einmal in der Liste auftreten dürfen (**sequence**)
- unique = true und ordered = false: es handelt sich um eine Menge (**set**)
- unique = false und ordered = true: es handelt sich um eine Liste im üblichen Sinne
- unique = false und ordered = false: es handelt sich um eine Kollektion, deren Elemente ungeordnet sind und mehrfach auftreten dürfen (**bag**)

Instanz- und Klassenattribute:

Attribute können für jede Instanz einer Klasse einen eigenen Wert besitzen oder aber für alle Instanzen der gegebenen Klasse einen eigenen Wert.

- alle bisher vorgestellten Attribute sind **Instanzattribute**
- **Klassenattribute** (in Java static genannt) werden unterstrichen

Beispiele:

- firstNames: String [0..3] = emptySet() {readonly, unique,ordered}

⇨

⇨

<u>+ /noOfClients : int = self.instances->size()</u>

⇨

⇨



Aufzählungen und andere primitive Typen als Klassen dargestellt:

<<enumeration>> Colour
black white red ...

<<primitive>> Name
prename: String surname: String

<<primitive>> String

- **Aufzählungen** (enumerations) werden wie normale Klassen mit einem besonderen Stereotyp <<enumeration>> dargestellt. Für die Definition der Literale der Aufzählung wird der Attributbereich missbraucht.
- **Einfache Datentypen** wie Integer oder Stringn werden als existent vorausgesetzt oder können als Klassen mit Stereotyp <<primitive>> eingeführt werden.
- **Datentypen mit interner Struktur** (Sätze, records) können als primitive Typen eingeführt werden; Attribute werden zur Definition der Komponenten verwendet.



Identifikation von Assoziationen:

Assoziationen sind (in aller Regel zweistellige) Relationen zwischen Klassen. Ihre Instanzen sind Links, die Instanzen der vorgebenen Klassen verbinden.

In erster Linie sind solche Assoziationen aufzuführen, die nicht nur temporär während der Abarbeitung einer Operation (Systemfunktion) existieren.

Kandidaten für Assoziationen:

- A ist ein logischer/physikalischer Teil von B (Client - Company)
- A überwacht/besitzt B (RentalOffice - MotorVehicle)
- A benutzt B (Client - MotorVehicle)
- A verweist auf B (InsuranceContract - RentalContract)
- A kommuniziert mit B (Client - Clerk)

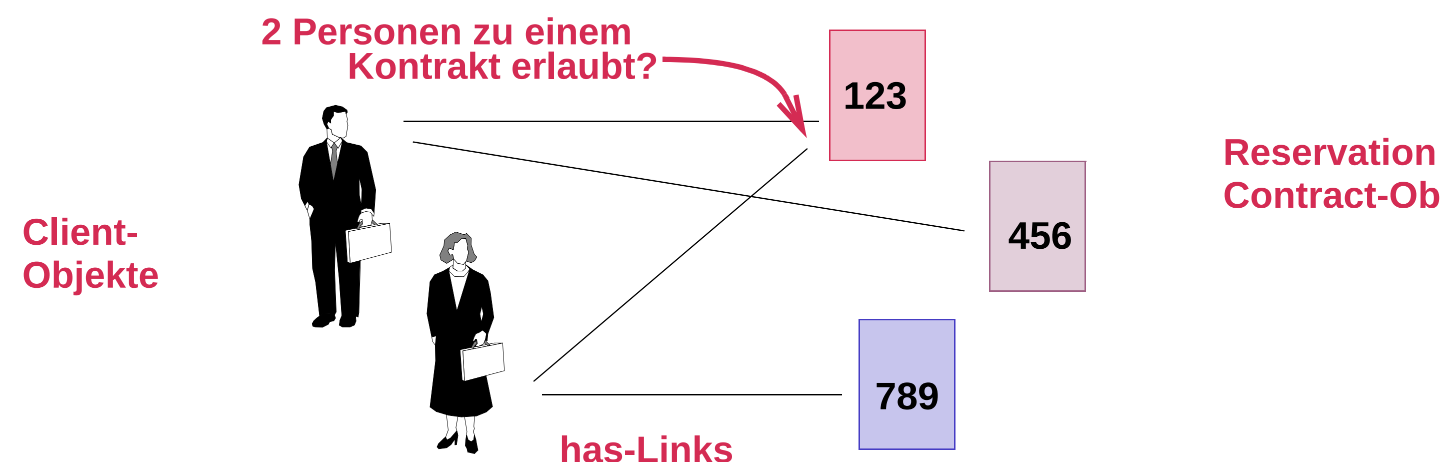


Eigenschaften von Assoziationen:

Beispiel:



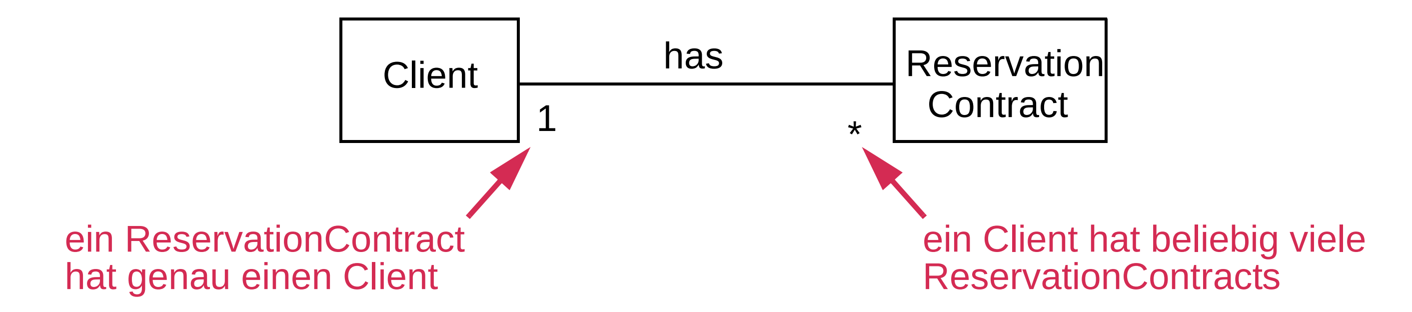
Fragestellung:





Eigenschaften von Assoziationen - 2:

- **Multiplizität** (Kardinalität):



Format von Multiplizitätsangaben: <untere Grenze> .. <obere Grenze>:

0 .. 1		1 .. *
1		1 .. 3
0 .. *	oder	*

- **Leserichtung**:

has(Contract) ▶

Client hat Contract und nicht umgekehrt



- **Navigierrichtung** (hat im Prinzip nichts mit Leserichtung zu tun):



Achtung:
Leserichtung = Navigationsrichtung meist sinnvoll

  - ⇨ normale Assoziationen sind **bidirektional** und können in beiden Richtungen traversiert werden
  - ⇨ **unidirektionale** Assoziationen (mit Pfeilspitze an einem Ende und Kreuz am anderen Ende) können nur in einer Richtung traversiert werden
  - ⇨ im Beispiel: Client verweist auf viele ReservationContracts, aber ein ReservationContract „kennt“ den zugehörigen Client nicht

- **Navigierrichtung** (diesmal verkehrt herum):



  - ⇨ jetzt „kennt“ ein ReservationContract den zugehörigen Client, aber ein Client „weiss nicht“, welche Reservierungen er hat
  - ⇨ Bedeutung von Assoziationsende ohne Pfeil und Kreuz nicht festgelegt.

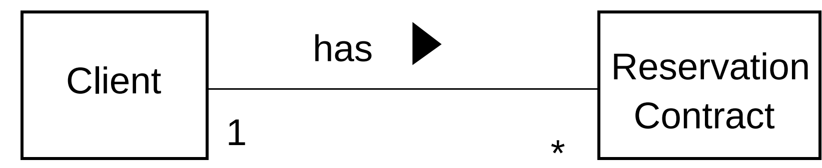


- **Navigierrichtung** - Abkürzungsregel 1:



⇨ wir interpretieren eine solche Assoziation als nur in einer Richtung navigierbar (Kreuz wird beim Client ergänzt)

- **Navigierrichtung** - Abkürzungsregel 2:



⇨ wir interpretieren eine solche Assoziation als in beiden Richtungen navigierbar (Pfeile werden auf Seiten beiden ergänzt)

- **Navigierrichtung** - diese Variante verwenden wir nicht:



⇨ beidseitig nicht navigierbare Assoziation als eigene Datenstruktur sinnvoll



Probleme mit Navigation bei Assoziationen auf einer Klasse:

Client 1 has ▶ * Reservation Contract
isBossOf 0..1 0..*

Klassendiagramm

Objektdiagramm

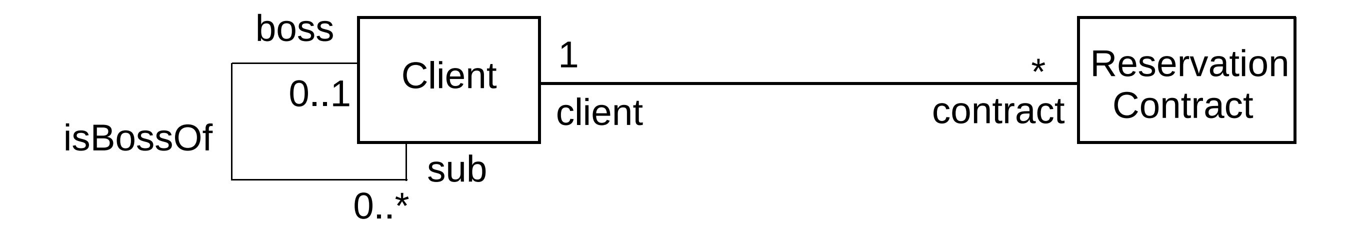
Margot:Client isBossOf Andy:Client
Margot:Client isBossOf Hannah:Client
Andy:Client isBossOf Tobias:Client
Andy:Client isBossOf Alexander:Client

Problem: wenn Leserichtung von isBossOf unklar ist, dann weiss man nicht, ob man mit lnkIsBossOf von Andy zu Margot oder zu Tobias und Alexander navigiert.



Eigenschaften von Assoziationen - 3:

- **Rollennamen** (Namen für Leserichtungen):
  - ⇨ Rollennamen beginnen immer mit einem Kleinbuchstaben
  - ⇨ Default-Wert für Rollenname ist Klassenname (z.B. client für Client)
  - ⇨ Assoziationen mit selber Start- und Zielklasse müssen Rollennamen haben
  - ⇨ Assoziationsname selbst darf fehlen (Default-Wert-Regel nicht definiert)

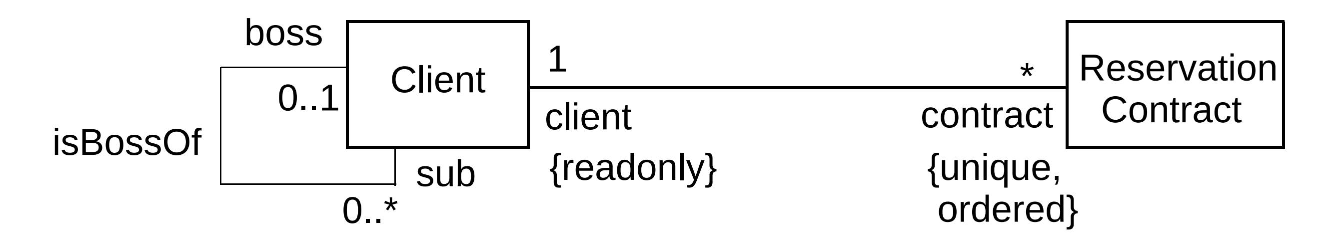


Rollennamen werden für den Zugriff auf benachbarte Objekte benötigt. Im folgenden Kapitel 7 werden wir sehen, wie Rollennamen innerhalb von prädikatenlogischen Ausdrücken und in der Implementierung von Klassen verwendet werden.



Eigenschaften von Assoziationsenden:

Die beiden Assozationsenden (Rollen) einer Assoziation können die gleichen Eigenschaften wie Attribute besitzen:



- **readonly**: Links dieser Assozation dürfen beim Objekt auf der „anderen“ Seite der „readonly“-Rolle nur bei der Initialisierung des Objektes angelegt und nur mit Objekt zusammen gelöscht werden
- **unique**: ist diese Eigenschaft an einem der beiden Assoziationsenden angegeben, so darf es keine „parallelen“ Links geben (also keine zwei Links zwischen dem selben Objektpaar
- **ordered**: aus Sicht des Objektes an der anderen Seite des betroffenen Assozations-endes ist die Menge der ausgehenden Links geordnet



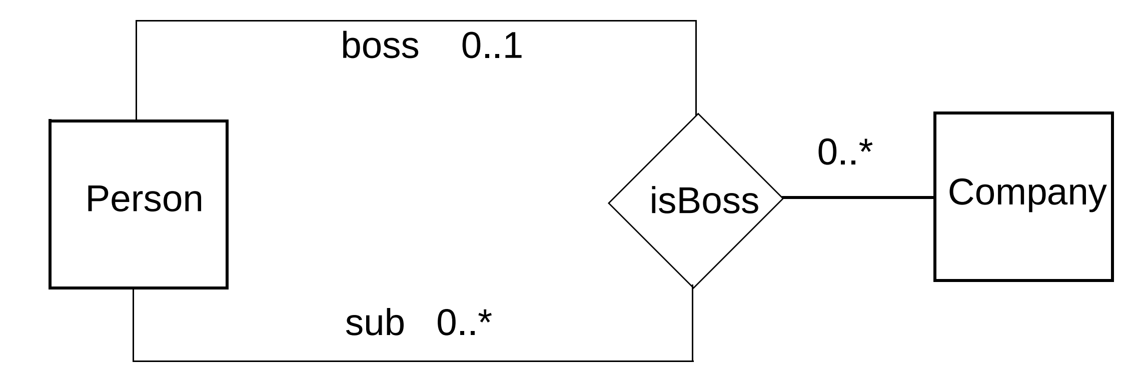
Kombinationen von unique und ordered:

- unique = true und ordered = true: es handelt sich um eine Liste von Elementen, die jeweils nur einmal in der Liste auftreten dürfen (**sequence**)
- unique = true und ordered = false: es handelt sich um eine Menge (**set**)
- unique = false und ordered = true: es handelt sich um eine Liste im üblichen Sinne
- unique = false und ordered = false: es handelt sich um eine Kollektion, deren Elemente ungeordnet sind und mehrfach auftreten dürfen (**bag**)



Von binären zu n-ären Assoziationen:

Sind Personen gleichzeitig Angestellte mehrerer Firmen, dann kann man mit binären „isBossOf“-Assozationen nicht ausdrücken, in welcher Firma eine Person Vorgesetzter einer anderen Person ist. Deshalb gibt es auch n-äre Assoziationen, die mehr als zwei Objekte verbinden.

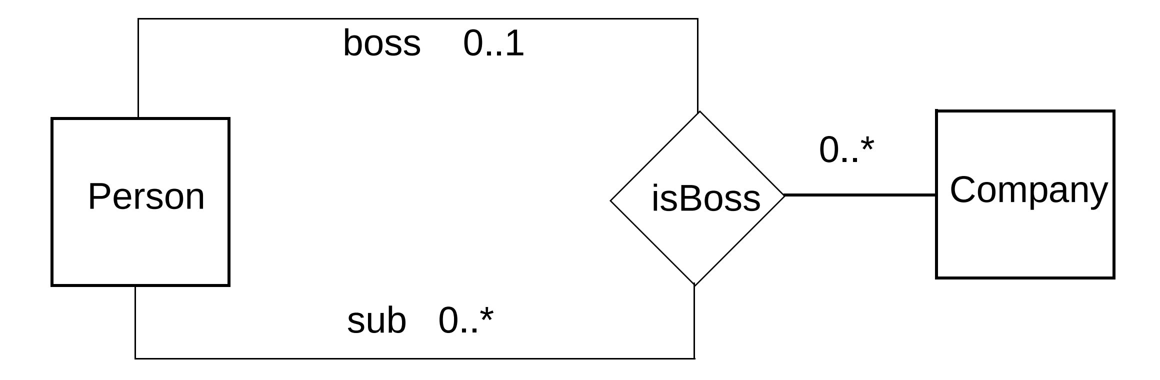


Semantik der Multiplizitäten n-ärer Assoziationen:

Wenn man n-1 Enden festhält (konkrete Objekte in einer Beziehung betrachtet), dann legt die Multiplizität des verbleibenden Endes fest, wieviele Links zu „Partnerobjekten“ es für die festgelegten Objekte geben kann.



Beispiel für die Semantik von Multiplizitäten für n-äre Assoziationen:

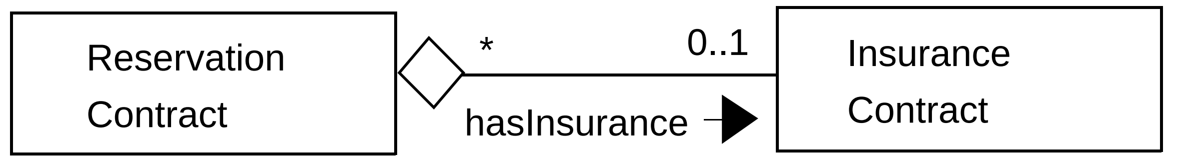


- eine Person in einer bestimmten Firma hat maximal einen direkten Vorgesetzten
- eine Person in einer bestimmten Firma hat beliebig viele Untergebene
- eine Person A kann in beliebig vielen Firmen Vorgesetzte einer Person B sein

Die hier gewählte Interpretation von Multiplizitäten ist vielleicht nicht auf den ersten Blick intuitiv, aber eine Verallgemeinerung des Falles n = 2 (binäre Relationen).

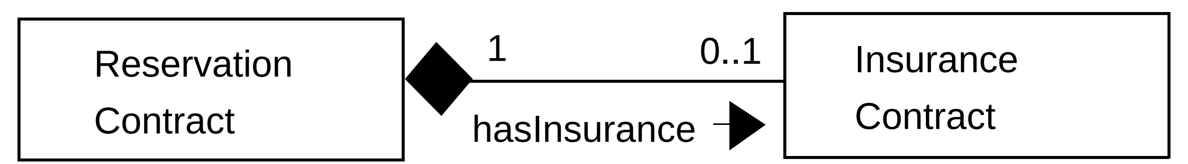


Aggregation:



- binäre Assoziation mit „vage“ definierter Zusatzsemantik (part-of-Beziehung)
- Aggregationsbeziehungen bilden **keine Zyklen** (Kreise)
- ein Objekt darf Teil mehrerer Elternobjekte (ReservationContracts) sein

Komposition:



- Spezialform der Aggregation, **propagiert Löschen** von Eltern zu Kindern (Löschen von ReservationContract löscht seinen InsuranceContract mit)
- Objekt darf **höchstens ein Elternobjekt** besitzen



Regeln für die Verwendung von Aggregation/Komposition:

Aggregation und Komposition müssen in der Analysephase nicht verwendet werden, da sie sich kaum von “normalen” Assoziationen unterscheiden.

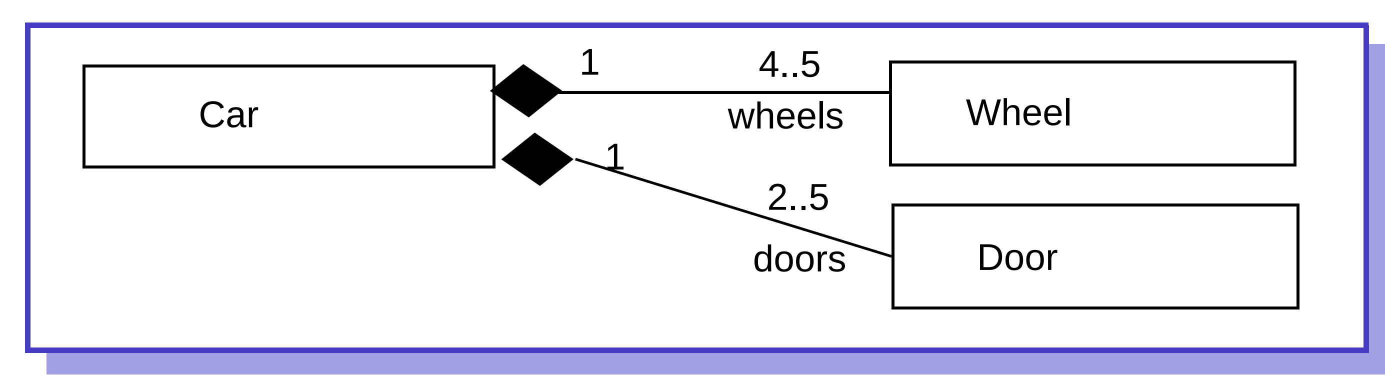
Einsatz von Aggregation/Komposition ggf. wenn:

- ein Objekt von anderem Objekt existenzabhängig ist (InsuranceContract von RentalContract)
- ein Objekt „offensichtlich” logischer/physikalischer Bestandteil eines anderen ist (Gesamtreservierungssystem enthält alle Daten)
- Eigenschaften des Elternobjekts sich auf die Kinder übertragen (Farbe eines Autos auf seine Karrosserie-Bestandteile)
- Operationen auf dem Elternobjekt zu den Kindern propagiert werden (so wie etwa löschen, bewegen, …)
- **im Zweifelsfall**: streitet man sich, ob eine Beziehung eine „normale“ Assoziation ist oder eine Aggregation bzw. Komposition, dann ist es eine Assoziation

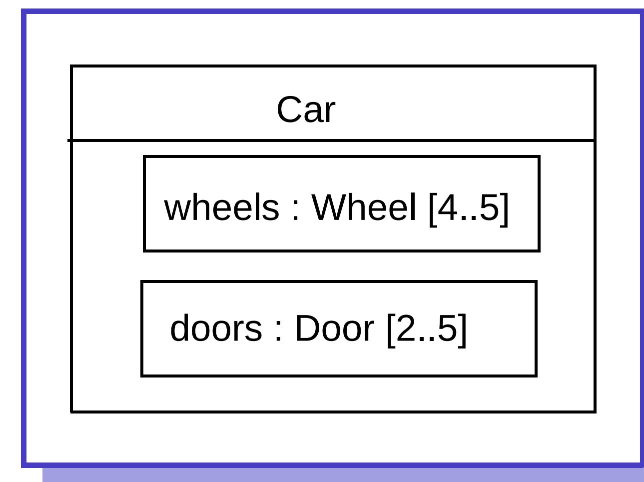


Verschiedene Darstellungen der Komposition:

a) normale UML-Variante



b) mit Schachtelung als Montagediagramm (Kompositionsstrukturdiagramm):



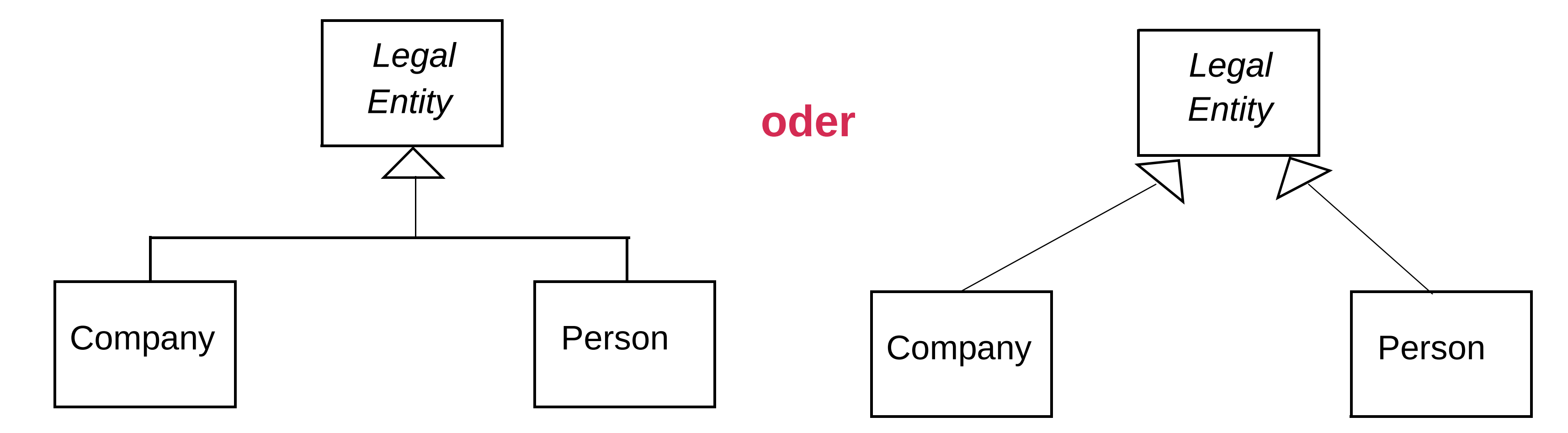
- jede Instanz der Klasse Car enthält 4 bis 5 Wheel-Instanzen, die über den Rollen-namen wheels angesprochen werden
- jede Instanz der Klasse Car enthält 2 bis 5 Door-Instanzen, die über den Rollenna-men doors angesprochen werden



Aufbau von Klassenhierarchien:

- jede Klasse kann im Allgemeinen mehrere **Oberklassen** besitzen
- besitzt sie höchstens eine Oberklasse, spricht man von **Einfachvererbung**, sonst von **Mehrfachvererbung**
- jede Klasse besitzt im Allgemeinen mehrere **Unterklassen**
- Unterklassen erben und erweitern Mengen von Attributen, Operationen, …

Beispiel für Einfachvererbung (single inheritance):





Definition von Instanz, Mitglied und Extension von Klassen:

- jedes Objekt ist **Instanz** genau einer (seiner) Klasse
- **abstrakte** Klassen (kursive Namen) besitzen keine Instanzen
- jedes Objekt ist **Mitglied**
  - ⇨ seiner eigenen Klasse
  - ⇨ und aller ihrer Oberklassen
- die **Extension** einer Klasse ist die Menge ihrer Mitglieder

Grafik für Extensionen:

Extension der Menge der LegalEntity-Objekte

Extension der Menge der Company-Objekte	Extension der Menge der Person-Objekte



Beispiel eines einfachen Klassendiagramms (ohne Attribute):

MVRS
Legal Entity
{disjoint}
Company
Person
{overlapping}
Rental Office
works For
Employee
Client
hasClients
hasVehicles
has Reservation Contracts
has Rental Contracts
Contract
{disjoint}
Reser-vation Contract
Rental Contract
Insurance Contract
hasInsurance
hasBill
Bill
hasCategory
Category
isOfCategory
reservedVehicle
rentedVehicle
Motor Vehicle
{overlapping}
Car
Truck
Van

MVRS = Motor Vehicle Reservation System



5.4 Exkurs zu Petri-Netzen

Petri-Netze wurden zur Modellierung und Analyse des Verhaltens von Systemen mit statischer Struktur und nebenläufig (parallel) arbeitenden Teilkomponenten entwickelt.

Ihre Urform wurde von **Adam Petri** in seiner Dissertation an der **TH Darmstadt** erfunden. Inzwischen gibt es eine fast unüberschaubare Vielzahl von Varianten.

Petri-Netze (in ihrer einfachsten Form) werden hier vorgestellt, da

- ⇨ sie eine geeignete Basis für die präzise Beschreibung der Semantik der UML-Aktivitätsdiagramme des nächsten Abschnitts darstellen
- ⇨ jeder Ingenieur zumindest einmal in seinem Leben diese Form der Modellierung von Systemen mit nebenläufigem (und ggf. auch nichtdeterministischem) Verhalten gesehen haben sollte
- ⇨ wegen des lokalen Bezugs zur TH/TU Darmstadt

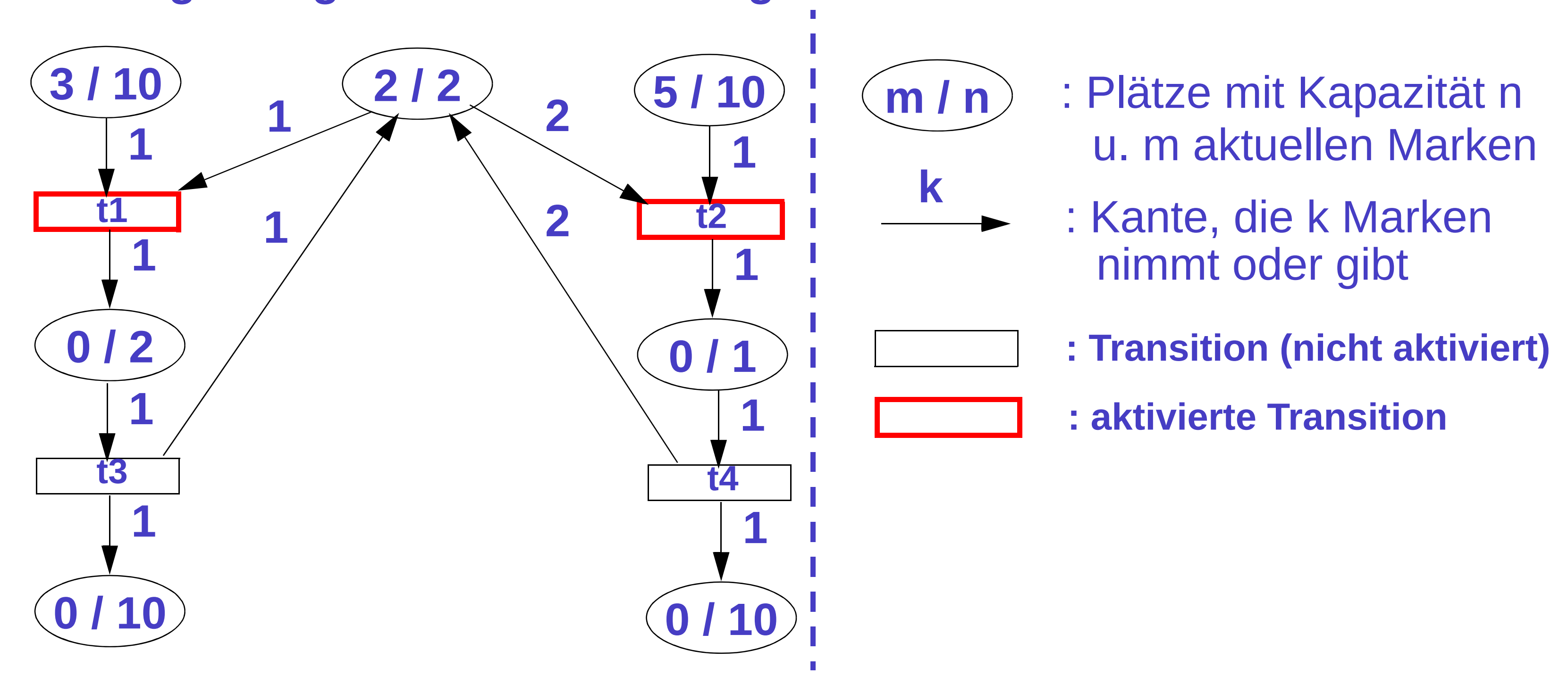


Grundkonzepte von Petri-Netzen:

- Ein Petri-Netz ist ein bipartiter (zweigeteilter) Graph mit zwei Arten von Knoten
  - ⇨ **Plätze/Stellen (Places)** speichern den Zustand eines verteilten Systems in Form von Marken (Token)
  - ⇨ **Transitionen (Transitions)** beschreiben Zustandsübergänge des Systems
- **Gerichtete Kanten (Arcs)** verbinden Plätze mit Transitionen und Transitionen mit Plätzen (aber niemals direkt Plätze mit Plätzen oder Transitionen mit Transitionen)
- Eine Transition kann **schalten** bzw. „feuern“ bzw. ist **aktiviert**, wenn
  - ⇨ auf allen ihren Vorplätzen (Plätze verbunden mit einlaufenden Kanten) genügend Marken liegen
  - ⇨ auf allen ihren Nachplätzen (Plätze verbunden mit auslaufenden Kanten) noch nicht zu viele Marken liegen
- Schaltet eine Transition, dann nimmt sie von allen ihren Vorplätzen Marken weg und fügt all ihren Nachplätzen Marken hinzu



Eine mögliche grafische Darstellung von Petri-Netzen:



- im aktuellen Zustand des Petri-Netzes können Transitionen t1 und t2 feuern
- wenn t1 gefeuert hat, dann haben Vorplätze nur noch 2 v. 10 bzw. 1 v. 2 Marken
- damit kann dann Transition t2 nicht mehr feuern, dafür aber t3 und nochmal t1
- ...



Konkretes Beispiel für Petri-Netz:

Verfügbare PKWs
6 / 10
1
Ausleihe
1
4 / 10
Ausgeliehene PKWs
1
Rückgabe
Tankgutscheine
96/100
1
2
1
1
0/100
Anzahl Rückgaben
Verfügbare Vans
5 / 5
1
1
Ausleihe
1
0 / 5
Ausgeliehene Vans
1
Rückgabe
1

Die Ausführung des Netzes terminiert, wenn

- ⇨ entweder keine Tankgutscheine mehr da sind
- ⇨ oder die Anzahl der Rückgaben die Zahl 100 erreicht hat



Analysen von Petri-Netzen (weitere Begriffe):

- Eine **Transition** t in einem Petri-Netz ist für eine Markierung m **tot**, wenn von m aus keine Markierung m´ erreicht werden kann, für die t aktiviert ist.
- Eine **Transition** t in einem Petri-Netz ist für eine Markierung m **lebendig**, wenn sie für keine von m aus erreichbare Markierung m´ tot ist; damit ist „lebendig" eine stärkere Forderung als „nicht tot":
  - ⇨ t ist für m nicht tot: es ist *irgendeine* Markierung von m aus erreichbar für die t aktiviert ist
  - ⇨ t ist lebendig für m: *für alle* von m aus erreichbaren Markierungen m´ gibt es eine von m´ erreichbare Markierung m´´, sodass t aktiviert ist
- Eine **Markierung** m eines Petri-Netzes ist **lebendig (tot)**, wenn *alle* seine Transitionen lebendig (tot) sind.
- Eine **Markierung** m ist **verklemmungsfrei**, wenn keine von m aus erreichbare Markierung m´ tot ist.
- Ein **Petri-Netz** ist **lebendig (tot, verklemmungsfrei)**, wenn seine Anfangsmarkierung lebendig (tot, verklemmungsfrei) ist.



Anmerkungen zu Petri-Netz-Eigenschaften:

- Sind mehrere Transitionen in einem Petri-Netz aktiviert, so feuert irgendeine davon (**nichtdeterministische Entscheidungen**). Die daraus resultierende Änderung der Belegung von Plätzen mit Marken kann dazu führen, dass bislang aktivierte **konkurrierende Transitionen** anschließend nicht mehr aktiviert sind.
- Ist eine Markierung m eines Petri-Netzes **lebendig**, dann gilt für jede mögliche Schaltreihenfolge und damit erreichbare Markierung m´ (ausgehend von m): keine Transition ist tot; es gibt also ausgehend von m´ für jede Transition eine Schaltreihenfolge, die diese Transition (wieder) aktiviert.
- Ist eine Markierung m eines Petri-Netzes **verklemmungsfrei**, dann gilt für jede mögliche Schaltreihenfolge und damit erreichbare Markierung m´ (ausgehend von m): es gibt immer mindestens eine aktivierte Transition
- Ist eine Markierung m eines Petri-Netzes **tot**, so ist keine seiner Transitionen aktiviert (und damit gibt es keine von m aus erreichbaren anderen Markierungen mehr).



Gegenbeispiel für Tote Transitionen im Petri-Netz:

Verfügbare PKWs
6 / 10
1
Ausleihe
1
4 / 10
Ausgeliehene PKWs
1
Rückgabe

Tankgutscheine
96/100
1
2

Verfügbare Vans
5 / 5
1
1
Ausleihe
1
0 / 5
Ausgeliehene Vans
1
Rückgabe

1
1
0/100
Anzahl Rückgaben

Für die obige Markierung ist **keine Transition tot**, da drei Transitionen direkt aktiviert sind und für die nicht aktivierte vierte Transition eine Schaltfolge existiert, sodass sie dann aktiviert ist.



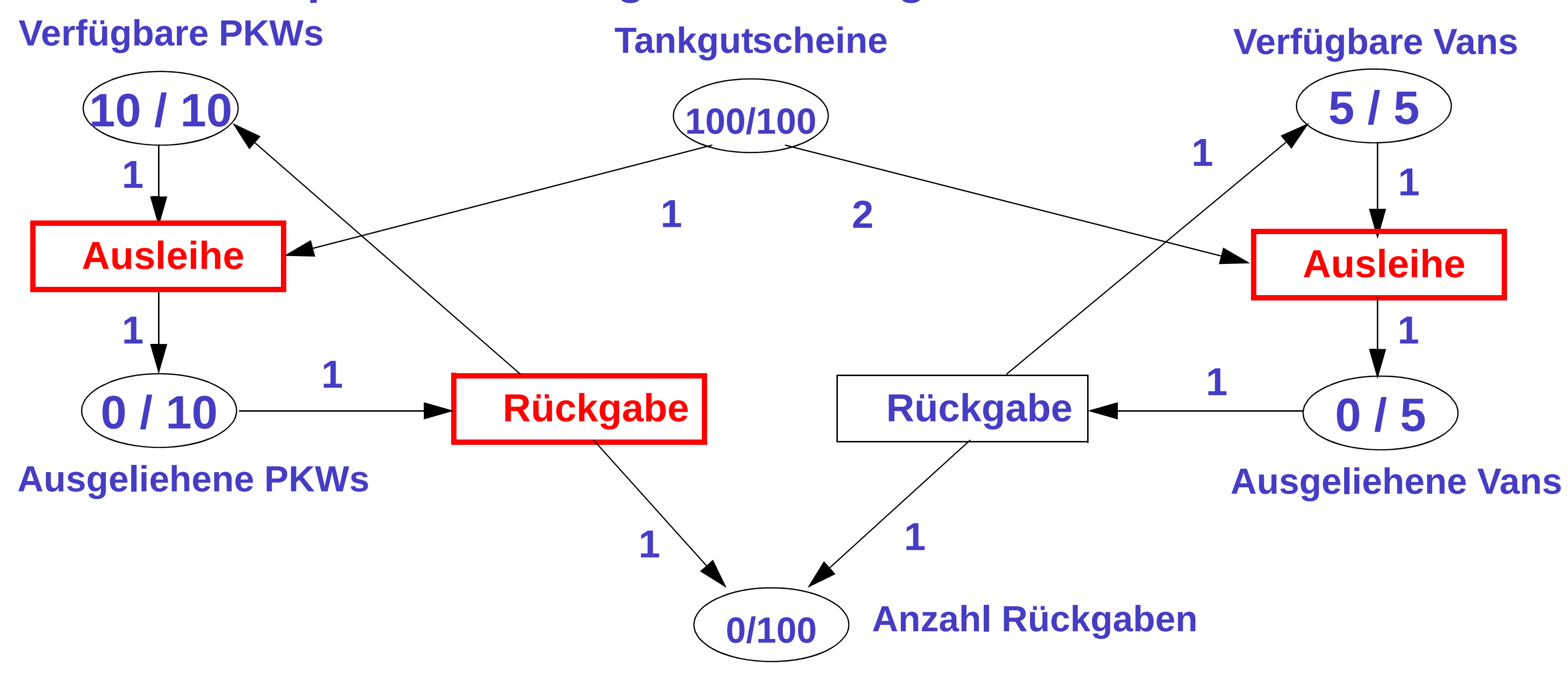
Gegenbeispiel Lebendige Transitionen im Petri-Netz:

Verfügbare PKWs
6 / 10
1
Ausleihe
1
4 / 10
Ausgeliehene PKWs
1
Rückgabe
Tankgutscheine
96/100
1
2
1
0/100
Anzahl Rückgaben
1
Rückgabe
Verfügbare Vans
5 / 5
1
1
Ausleihe
1
1
0 / 5
Ausgeliehene Vans

Für die obige Markierung (und jede beliebige andere Markierung) ist **keine Transition lebendig**, da für jede der vier Transitionen eine Schaltfolge existiert, sodass danach die Transition nie mehr aktiviert werden kann.



Petri-Netz-Beispiel mit Anfangsmarkierung:



Das obige Petri-Netz mit der vorgegebenen Anfangsmarkierung ist weder lebendig noch verklemmungsfrei noch tot. Es gilt sogar, dass das Petri-Netz mit keiner möglichen Anfangsmarkierung lebendig oder verklemmungsfrei ist, da jede mögliche Ausführung nach einer endlichen Anzahl von Schritten terminiert (da keine Transition mehr aktiviert ist).



Petri-Netz-Varianten und -Werkzeuge:

- die hier vorgestellten Petri-Netze sind sehr „ausdrucksschwach" und eignen sich kaum für die Modellierung realer Anwendungen. Deshalb gibt es viele Vorschläge wie man den Petri-Netz-Formalismus erweitern kann:
  - **Gefärbte Petri-Netze:** die Marken sind Werte bestimmter Typen mit denen gerechnet werden kann (Zahlen, Strings, ...)
  - **Zeitbehaftete Petri-Netze**: dem Verharren von Marken auf Plätzen und/oder dem Schalten von Transitionen werden Zeiten zugeordnet
  - **Hierarchische Petri-Netze**: erlauben eine strukturierte übersichtlichere Darstellung von großen Netzen durch Verfeinerung von Plätzen und/oder Transitionen
  - **Prädikat-/Transitionsnetze:** Transitionen können mit Prädikaten anno-tiert werden, die das Schalten von Transitionen erlauben/verhindern
- Eine Übersicht über Petri-Netz-Werkzeuge findet man z.B. unter: http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/; ein einfaches Applet ist http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/java/Guth/



5.5 Ablaufmodellierung (UML Activity Charts)

Aufgabe:

Die Beschreibung von Systemfunktionen beschränkte sich bislang auf die exemplarische Modellierung einzelner Geschäftsvorfälle bzw. Szenarien als Anwendungsfälle.

Vorgehensweise:

Mit Aktivitätsdiagrammen (Activity Charts)

- ⇨ wird der **zeitliche Zusammenhang** einzelner **Geschäftsvorfälle** in Form von Geschäftsprozessen modelliert
- ⇨ werden **alle Alternativen** einer Systemfunktion gleichzeitig erfasst
- ⇨ wird erstes Augenmerk auf **parallel durchführbare** Aktionen gerichtet
- ⇨ wird erster **Zusammenhang** zwischen Objekten und Aktionen geschaffen



Vorbemerkungen zu den Aktivitätsdiagrammen:

- Aktivitätsdiagramme sind die jüngste Diagrammart der UML-Familie (und wurden in UML 2.0 komplett neu definiert)
- sie sind ein Hybrid aus
  - den „alten” Datenflussdiagrammen (siehe Abschnitt 4.1, Seite 110)
  - den hoffentlich aus dem Studium bekannten Kontrollflussdiagrammen
  - den gerade vorgestellten Petri-Netzen
- sie werden
  - sowohl als visuelle Programmiersprache für die Implementierung einzelner Operationen von Klassen
  - als auch zur Modellierung von Geschäftsprozessen eingesetzt (Zusammenhang von Anwendungsfällen und Detailierung)
- es handelt sich also um eine prinzipiell **ausführbare Diagrammart** (im Gegensatz zu den bisherigen Diagrammarten)



Aktivitätsdiagramm MakeReservation für einfachen Anwendungsfall:

MakeReservation

Aktivität

Anfang

Kontroll-
fluss

Aktion

FindClientInDB

FindVehicles

determines set of all
vehicles vs of category c

SelectFreeVehicle

Selects free vehicle v for
given period p that is
not reserved during p

CreateReservation

CreateMessage

Ende



Aktivitätsdiagramm MakeReservation mit Fallunterscheidung:

MakeReservation

[client not in DB]

FindClientInDB

[client in DB]

FindVehicles

determines set of all vehicles vs of category c

SelectFreeVehicle

Selects free vehicle v for given period p that is not reserved during p

CreateReservation

[free vehicle found]

CreateMessage

[no free vehicle]

Fallunterscheidung

Bedingungen (sollten sich gegenseitig ausschließen)



Erläuterungen zu einfachen Aktivitätsdiagrammen:

- einfache Aktivitätsdiagramme entsprechen den „alten“ Kontrollflussdiagrammen
- Aktionsknoten können beliebige Anweisungen (Texte) enthalten; dazu gehört der Aufruf eines anderen Aktivitätsdiagramms
- die Ausführung beginnt mit der Ablage einer Marke (Token) am Anfangsknoten
- jeder Ausführungsschritt setzt die Marke entlang einer Kontrollflusskante zum nächsten Knoten
- an einem Bedingungsknoten (Raute) wird die Marke entlang einer Kontrollflusskante propagiert, deren Bedingung wahr ist (Text in [...]).
- es sollte immer nur die Bedingung einer Kante wahr sein, die aus einem betrachteten Bedingungsknoten ausläuft
- die Ausführung eines aufgerufenen Aktivitätsdiagramms ist beendet, wenn eine Marke den Endknoten erreicht
- mehrere gleichzeitige Ausführungen / Aktivierungen eines Aktivitätsdiagramms sind im Normalfall nicht zulässig (UML-Konstrukt hierfür wird nicht vorgestellt)



Aktivitätsdiagramm MakeReservation mit Objektfluss:



Erläuterungen zu Aktivitätsdiagrammen mit Objektfluss:

- es handelt sich dabei um moderne Variante der Datenflussdiagramme
- eine Kontrollflusskante wird durch eine oder mehrere Objektflusskanten ersetzt, die Ausgaben einer Aktion zur nächsten Aktion weitergeben
- statt Kontrollmarken (control tokens) entlang Kontrollflusskanten zu verschieben, werden Objekte (data tokens) entlang von Objektflusskanten verschoben
- Ausführung eines Aktivitätsdiagramms beginnt, wenn auf allen Eingabe-parameterplätzen mindestens ein Objekt liegt (im einfachsten Fall); die Objekte können dort in einer Warteschlange abgelegt werden
- einmalige Ausführung konsumiert von jedem Eingabeparameterplatz genau ein Objekt
- Ausführung endet damit, dass auf alle Ausgabeparameterplätze neue Objekte gelegt werden (im einfachsten Fall) und/oder Endeknoten erreicht wurde; die Objekte können dort in einer Warteschlange abgelegt werden



Erläuterungen zu Aktivitätsdiagrammen - Fortsetzung:

- stellt eine Aktion einen Aufruf eines Aktivitätsdiagramms dar, so entsprechen
  - die Eingabeparameter der Aktivität den Eingabe-Pins der Aktion
  - die Ausgabeparameter der Aktivität den Ausgabe-Pins der Aktion
- UML kennt eine eigene Teilsprache für die Programmierung einfacher Aktionen, die **Action Language**, folgendes unterstützt:
  - Erzeugen/Löschen von Objekten
  - Erzeugen/Löschen von Links
  - Abfragen auf Objekten und Links
  - ...
- das Weitergeben von mehreren **Kopien von Objekten** an verschiedene Aktionen wie bei period wird eigentlich umständlicher modelliert (durch Verzweigungsknoten, siehe Seite 238 und Seite 239)
- viele weitere Details können noch geregelt werden - werden aber nicht von allen Werkzeugen unterstützt



Datenspeicherung in Aktivitätsdiagrammen:

MakeReservation

clientNo
cn
FindClientInDB
<<selection>>
cl.name = cn
<<datastore>>
ClientDB
cl
cl
cl
c
category
FindVehicles
<<selection>>
vs = collection of v with
v.category = c
<<datastore>>
VehicleDB
vs
vs
vs
period
SelectFreeVehicle
<<selection>>
rs = collection of r with
r.vehicle = v
P
v
ps
<<datastore>>
ReservationDB
v
p
CreateReservation
r
r
CreateMessage
m
message

Speicher (Datenbank) für Objektmengen (oft nur einer bestimmten Klasse)



Erläuterungen zu Datenspeicherung in Aktivitätsdiagrammen:

- Aktivitätsparameter und Pins besitzen i.a. die Fähigkeit, Objekte zwischenzuspeichern (Kapazität kann definiert werden) und in definierbarer Reihenfolge weiterzugeben - aber immer nur an eine Aktion/Aktivität
- so genannte **datastores** (Datenspeicher) können größere Objektmengen persistent (dauerhaft) speichern; auf sie kann in mehreren Diagrammen von mehreren Aktionen aus zugegriffen werden
- so genannte **centralBuffer** (temporäre Datenspeicher) können größere Objektmengen temporär zwischenspeichern; auch auf sie kann in mehreren Diagrammen von mehreren Aktionen aus zugegriffen werden (nicht gezeigt)
- mit Selektionsbedingungen (**selections**) in Form von Kommentaren an Objektflüssen kann definiert werden, welche Objekte aus einem datastore oder aus einem centralBuffer herausgeholt werden sollen



Alternative Darstellungen von Fallunterscheidungen:

MakeReservation

clientNo
cn
FindClientInDB
cl
<<datastore>>
ClientDB
cl
[cl == null]
cl
[cl != null]
category
FindVehicles
vs
<<datastore>>
VehicleDB
c
vs
vs
e
[error == true]
CreateError
period
SelectFreeVehicle
ps
m
P
v
<<datastore>>
ReservationDB
[v != null]
v
e
p
CreateReservation
HandleOverbooked
r
m
r
CreateMessage
message
m

Kontrollfluss ist Spezialfall von Datenfluss mit Dummy-Token-Werten

bedingter Kontrollfluss

impliziter Kontrollfluss

bedingter Objektfluss

lesender u. schreibender Zugriff auf Speicher

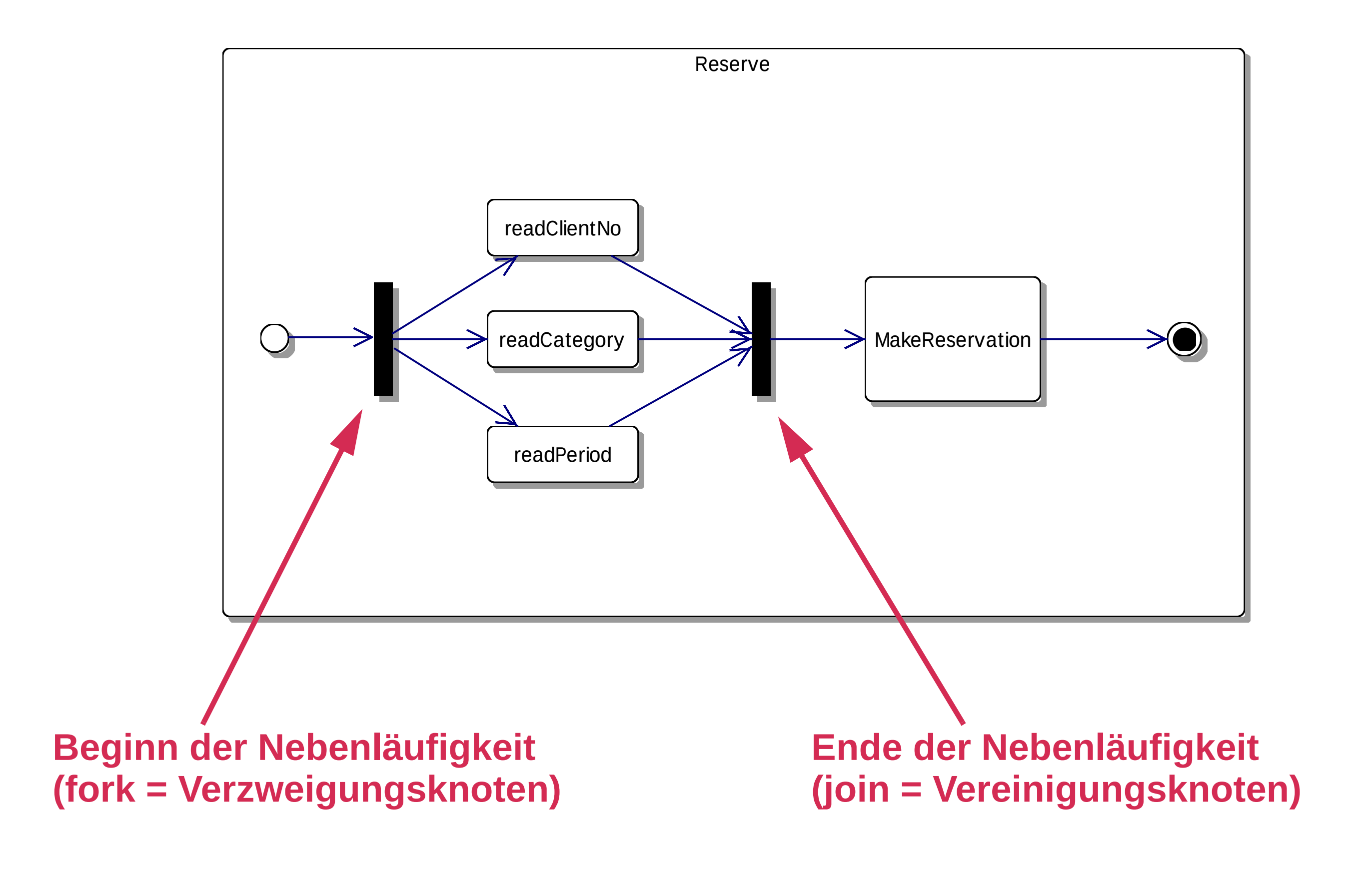


Kommentare zu vorigem Aktivitätsdiagramm:

- die Erhebung der Reservierungsdaten clientNo, category und period muss bereits vor der Durchführung der Aktivität MakeReservation erfolgt sein
- mit den bisherigen Sprachmitteln müssten wir genau eine Reihenfolge des Einlesens/Abfragens der Reservierungsdaten festlegen
- die Behandlung von Sonderfällen wird in Form von Fallunterscheidungen in den normalen Ablauf eingebaut (entgegen Ratschläge in Abschnitt 5.2)
- die Beschreibung einzelner Anwendungsfälle mit Aktivitätsdiagrammen hat den Vorteil, dass sie präziser als Fließtext sind
- hauptsächlich werden heute aber Aktivitätsdiagramme für die Beschreibung des Zusammenspiels (der zeitlichen Abfolge) einzelner Anwendungsfälle benutzt
- die Aktionen (Prozeduren) in den einzelnen Aktivitätsknoten sind noch keinen Klassen zugeordnet (diese fehlen zum großen Teil noch)



Modellierung nebenläufiger Aktivitäten:

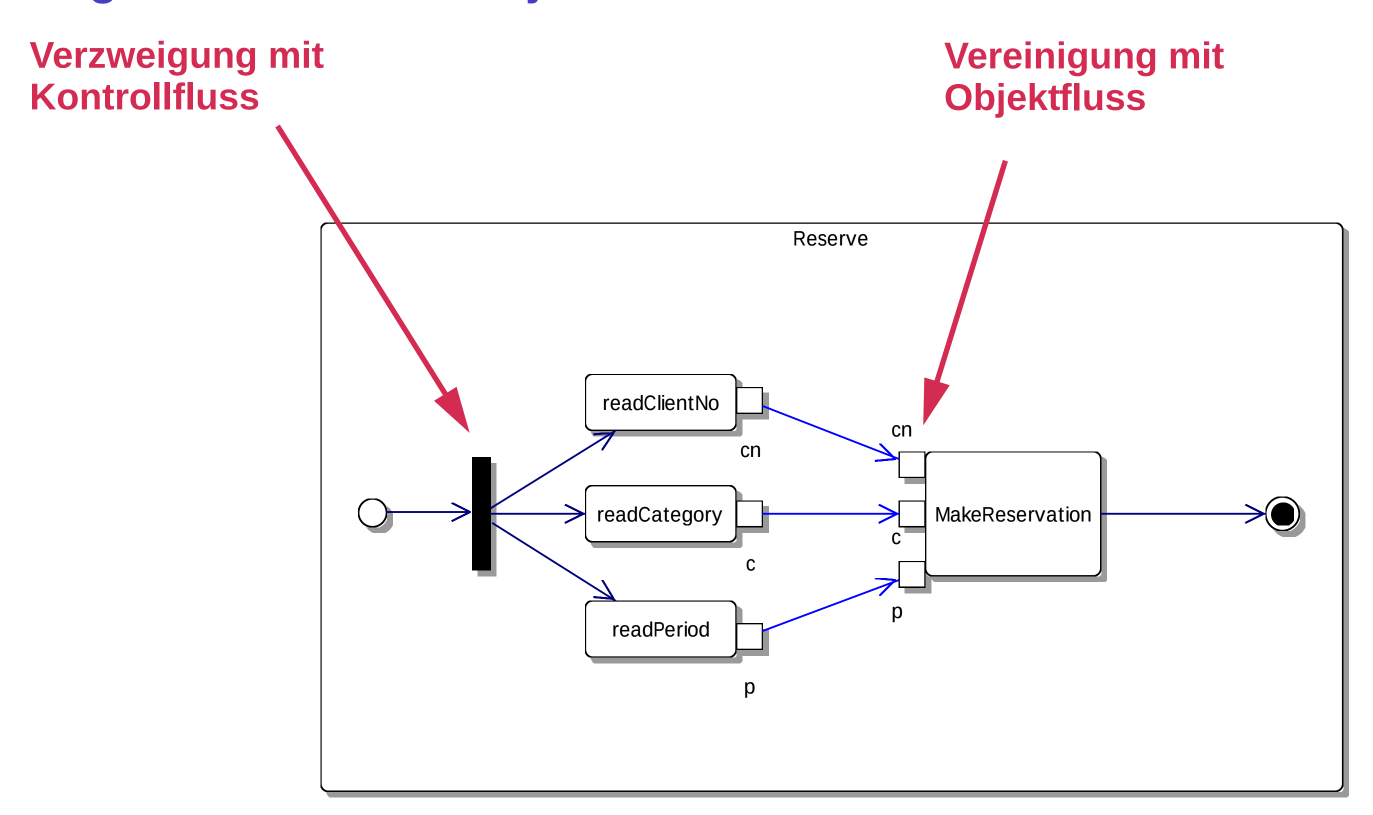


Beginn der Nebenläufigkeit (fork = Verzweigungsknoten)

Ende der Nebenläufigkeit (join = Vereinigungsknoten)



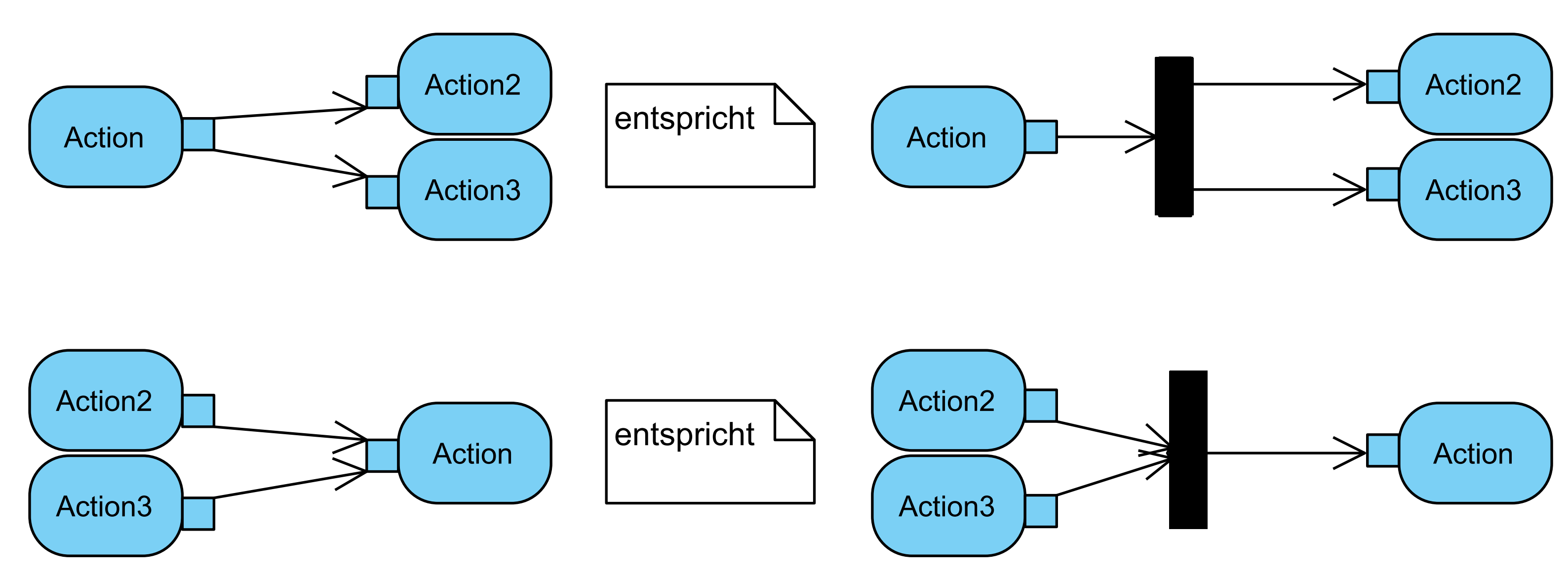
Nebenläufige Aktionen mit Objektfluss:



Auch die umgekehrte Situation ist denkbar: Verzweigung mit Objektfluss ohne fork-Knoten wie etwa im Beispiel auf Seite 231 gezeigt sowie Vereinigung von Kontrollflüssen mit join-Knoten



Schreibabkürzungen für Objektflussverzweigung und - zusammenführung:



Mehrere auslaufende oder einlaufende Objektflüsse haben also eine „und"-Semantik. Es wird bei mehreren auslaufenden Datenflüssen das von Action produzierte Token (Wert) kopiert (wie bei fork) und an Action2 und Action3 weitergeleitet. Bei mehreren einlaufenden Datenflüssen werden die ankommenden Werte „irgendwie" verschmolzen und zusammen weitergeleitet (wie bei join).



Aktivitätsdiagramm zur Beschreibung des Gesamtprozesses:

Ausnahmen selbst und Bereiche, die durch Ausnahmen verlassen werden können, werden normalerweise etwas anders dargestellt.



Erläuterungen zu Übersichts-Aktivitätsdiagrammen mit Ereignissen:

- es werden nicht Abläufe einzelner Systemfunktionen beschrieben, sondern es wird skizziert, wie alle Funktionen des Systems zusammenspielen
- Übergänge (Transitionen) zwischen Aktionen werden oft durch externe Ereignisse ausgelöst wie fetchEvent (Kunde kommt ins Büro, um Auto abzuholen)
- Ablauf einer bestimmten Zeitspanne (Timeout) kann auch ein Ereignis sein, das die Ausführung einer nachfolgenden Aktion auslöst
- <<exception>> Transitionen brechen den normalen Ablauf in einem Bereich ab und führen zur sofortigen Ausführung der Zielaktion
- Übergänge zwischen Aktionen können auch Ereignisse auslösen wie hier SendAlertMessage, das woanders wieder eine Transition auslöst
- gehen aus einer Aktion mehrere Kontrollflusskanten zu Ereignissen aus, dann „feuert“ die Kante, deren Ereignis zuerst eintritt
- „Swimlanes“ (Bereiche) werden zur übersichtlicheren Strukturierung der Diagramme verwendet und zur Definition eines „Wirkungsbereiches“ für Ausnahmen



Zusammenfassung für Aktivitätsdiagramme:

Aktivitätsbereich

Anfang

Beginn Fallunterscheidung

Aktion3

Ende Verzweigung

Aktion

[Nicht Bedingung]

[Bedingung]

Aktion4

Beginn
Nebenläufigkeit

<<datastore>>
persistente Datenspeicherung

Ende

Aktion2

Ereignis wird empfangen

Ereignis wird weggeschickt

Pin

Parameter

<<centralBuffer>>
temporäre Datenspeicherung

Ende der
Nebenläiufigkeit

Zeit-Ereignis

nur dieser Teilablauf wird beendet; andere
nebenläufige Aktionen werden fortgeführt

Immer noch unvollständige Übersicht über alle Elemente von Aktivitätsdiagrammen.



Abschließende Anmerkungen zu Aktivitätsdiagrammen:

- es handelt sich um die Diagrammart, die in UML 2.0 am radikalsten überarbeitet wurde
- eine ganze Reihe von Modellierungselementen (für mengenwertige Pins, loops, ...) wurden hier nicht vorgestellt
- viel Energie ist in die präzise Definition der Semantik dieser Diagramme geflossen; trotzdem ist eine Reihe von Punkten nicht geklärt
- Theoretisch kann mit Aktivitätsdiagrammen „programmiert" werden, Codegeneratoren dafür sind aber kaum verfügbar

Einsatzmöglichkeiten:

- **üblich**: präzise(re) Definition einzelner Anwendungsfälle oder des Zusammen-spiels von Anwendungsfällen (Geschäftsprozessmodellierung)
- **unüblich**: Programmierung reaktiver Systeme unter Einsatz von Ereignissen; hierfür werden eigentlich immer später vorgestellte Statecharts verwendet



5.6 Aufbau und Funktion eines Pflichtenheftes

Der **IEEE/ANSI-Standard 830-1993** schlägt in etwa folgenden Aufbau für Pflichtenheft vor (insgesamt 31 Seiten Text):

1. Einleitung mit
   - ⇨ Ziel des Anforderungsdokumentes (purpose)
   - ⇨ Anwendungsbereich des Produkts (scope)
   - ⇨ Definitionen, Akronyme, Abkürzungen (definitions)
   - ⇨ Referenzen auf andere Quellen (references)
   - ⇨ Überblick über Rest des Dokumentes (overview)
2. Allgemeine Beschreibung
   - ⇨ Produktionfunktionen, Benutzercharakteristika etc.
3. Spezifische Anforderungen:
   - ⇨ (nicht-)funktionale Anforderungen, Schnittstellenbeschreibungen etc.
4. Anhänge (mit Index)



Aufbau eines Pflichtenheftes nach [Ba96]:

Aus [Ba96] übernehmen wir einen Vorschlag für den Aufbau eines Pflichtenheftes, der

- das Pflichtenheft als **Erweiterung des Lastenheftes** ansieht, also die in Abschnitt 3.3 vorgeschlagene Gliederung eines Lastenheftes erweitert
- (fast) alle wichtigen Punkte der IEEE/ANSI-Norm in etwas anderer Anordnung übernimmt
- sich auf die Beschreibung der Software eines Gesamtsystems konzentriert
- der vorerst die Beschreibung von Testfällen fehlt (siehe Kapitel 8)
- und von uns optional um eine **Einleitung** ergänzt wird, die
  - die erwartete Leserschaft des Pflichtenheftes festlegt
  - die Versionsgeschichte des Pflichtenheftes erläutert
  - auf andere relevante Dokumente wie das Lastenheft verweist
  - und den Aufbau des Pflichtenheftes beschreibt



Aufbau eines Pflichtenheftes - 1:

1. **Einleitung** (neu, war für „kompaktes“ Lastenheft nicht unbedingt notwendig):

   … (siehe vorige Folie)

2. **Zielbestimmung** (Verfeinerung des entsprechenden Lastenheftkapitels):

   Das „Warum“ steht im Vordergrund, es wird in Form zu erreichender Ziele (Kriterien für die Softwarefunktionalität) festgehalten:

   ⇨ **Musskriterien**: unbedingt zu erreichende Ziele

   ⇨ **Wunschkriterien**: nicht unabdingbare, aber sehr wünschenswerte Ziele

   ⇨ **Abgrenzungskriterien**: was soll mit der Software **nicht** erreicht werden

3. **Produkteinsatz** (Verfeinerung des entsprechenden Lastenheftkapitels):

   ⇨ **Anwendungsbereiche**: Aufgabenfelder, die unterstützt werden

   ⇨ **Zielgruppen**: „Stakeholders“, die mit Softwaresystem umgehen werden

   ⇨ **Betriebsbedingungen**: wo und unter welchen Randbedingungen wird die Software eingesetzt (Büro, mobiler Einsatz, …)



Aufbau eines Pflichtenheftes - 2:

4. **Produktübersicht** (neuer Abschnitt):

   Gibt einen Überblick über das Produkt (seine Funktionen) sowie seine Rolle in allen relevanten Geschäftsprozessen (Verarbeitungsprozessen).

   Neben Fließtext werden hauptsächlich folgende UML-Diagrammarten eingesetzt:

   ⇨ **Aktivitätsdiagramme** für die Beschreibung von Geschäftsprozessen (Aktivitätsbereiche werden für die Zuordnung von Aktivitäten/System-funktionen zu Anwendungsbereichen verwendet)

   ⇨ **Anwendungsfall-Paketdiagramme** für die Unterteilung von Pro-duktfunktionen in Gruppen (orientiert an Anwendungsbereichen etc.)

   ⇨ **Anwendungsfall-Diagramme** mit Hauptfunktionen des Produkts als „primäre" Anwendungsfälle und den Zielgruppen als Akteure (plus andere Teilsysteme, Sensoren, etc. bei eingebetteten Systemen)



Aufbau eines Pflichtenheftes - 3:

5. **Produktfunktionen** (Verfeinerung des entsprechenden Lastenheftkapitels):

   Wie im Lastenheft durchnummeriert werden alle Produktfunktionen hier detaillierter beschrieben (mit Verweis auf damit umgesetzte Muss- oder Wunschkriterien):

   - ⇨ die Gliederung (Paketstruktur) wird aus der Produktübersicht übernommen und ggf. verfeinert
   - ⇨ jede Hauptfunktion (primärer Anwendungsfall) wird mit Hilfe eines Textschemas beschrieben (Verweise auf Glossar!)
   - ⇨ Spezialfälle oder oft benötigte Hilfsfunktionen werden als „sekundäre" Anwendungsfälle wie in Abschnitt 5.2 vorgeschlagen ausgelagert
   - ⇨ der Zusammenhang von primären und sekundären Anwendungsfällen kann durch weitere Anwendungsfalldiagramme festgehalten werden
   - ⇨ für eine präzisere Beschreibung von Anwendungsfällen können in Einzelfällen Aktivitätsdiagramme eingesetzt werden
   - ⇨ ggf. werden auch die erst in Abschnitt 7.3 eingeführten Sequenzdiagramme dafür verwendet (insbesondere wenn zeitliche Aspekte wichtig sind)



Aufbau eines Pflichtenheftes - 4:

6. **Produktdaten** (Verfeinerung des entsprechenden Lastenheftkapitels):

   Die längerfristig zu speichernden Daten des Systems werden - ggf. wie im Lastenheft durchnummeriert - aus Anwendersicht beschrieben (konzeptuelles Datenmodell). Dabei werden die Daten (mit Mengenangaben) entweder

   ⇨ rein textuell beschrieben wie im Lastenheft (wieder Verweise auf Glossar!)

   ⇨ oder in Form von UML-Klassendiagrammen mit zusätzlichen Kommentaren erfasst (einfache Variante im Sinne von Abschnitt 5.3)

   Achtung: soll ein sogenanntes „ausführbares“ Pflichtenheft mit einem „Rapid Prototyp“ des Softwaresystems erstellt werden, so muss

   ⇨ ein feineres Klassenmodell im Sinne von Abschnitt 7.1 erstellt werden

   ⇨ ggf. das Zusammenspiel der Operationen verschiedener Klassen durch die in Abschnitt 7.3 eingeführten Interaktionsdiagramme beschrieben werden

   ⇨ zu einigen Klassen eine Beschreibung ihres isolierten Verhaltens durch die in Abschnitt 7.4 eingeführten Statecharts (Automaten) hinzugefügt werden



Aufbau eines Pflichtenheftes - 5:

7. **Produktleistungen** (Verfeinerung des entsprechenden Lastenheftkapitels):

   Weitere Angaben zu den einzelnen Produktfunktionen oder Produktdaten der vorangegangenen Kapitel. Hier werden oft Leistungsanforderungen bzgl. Zeit und Genauigkeit angegeben. Verzichtet man auf diesen Abschnitt nicht ganz, dann wird man ggf. hier bereits die Interaktionsdiagramme und Statecharts der UML verwenden (siehe Kapitel 7).

8. **Qualitätsanforderungen** (Verfeinerung des entsprechenden Lastenheftkapitels):

   Wieder wird für die in Abschnitt 1.4 eingeführten Softwarequalitätsmerkmale in Matrix-Form angegeben, wie wichtig sie sind (siehe auch DIN ISO Norm 9126 zu Qualitätsanforderungen an Software).

9. **Benutzungsoberfläche** (neuer Abschnitt):

   Grundlegende Anforderungen an die Benutzeroberfläche (wie Gestaltungsricht-linien) werden hier festgehalten; zu einem **ausführbaren Pflichtenheft** gehört auch ein „Rapid Prototype“ der tatsächlichen späteren Benutzeroberfläche.



Aufbau eines Pflichtenheftes - 6:

10. **Nichtfunktionale Anforderungen** (neuer Abschnitt):

    Alle in den bisherigen Kapiteln nicht unterzubringenden Anforderungen werden hier aufgeführt (einzuhaltende Gesetze, Normen, Sicherheitsanforderungen, …).

11. **Technische Produktumgebung** (neuer Abschnitt):

    Die Umgebung wird beschrieben, in der das zu erstellende Produkt eingesetzt wird. Dabei wird wie folgt unterteilt:

    ⇨ **Hardware** (auf der Produkt läuft): meist werden zur Beschreibung entweder Fließtext oder diverse Diagrammarten wie Datenfluss-Diagramme (s. Seite 110) oder UML-Deployment-Diagramme (siehe Kapitel 7) eingesetzt

    ⇨ **Software** (die Produkt voraussetzt mit Beschreibung von Schnittstellen): meist werden zur Beschreibung Fließtext oder UML-Klassendiagramme eingesetzt, ggf. auch UML-Komponentendiagramme (s. Kapitel 7)

    ⇨ **Orgware**: organisatorische Randbedingungen, die erfüllt sein müssen

    Achtung: bei eingebetteten Systemen direkt nach Kapitel 3 oder Kapitel 4!



Aufbau eines Pflichtenheftes - 7:

12. **Entwicklungsumgebung** (neuer Abschnitt):

    Die Umgebung wird beschrieben, in der das zu erstellende Produkt entwickelt wird (Entwicklungsplattform). Insbesondere bei eingebetteten Systemen unterscheidet sich die Entwicklungsplattform sehr deutlich von der Zielplattform. Das Kapitel ist wie das vorangegangene Kapitel aufgebaut.

13. **Gliederung in Teilprodukte** (neuer Abschnitt):

    Für die iterative Erstellung des Produktes (Gesamtfunktionalität wird über mehrere Releases hinweg „stückweise“ zur Verfügung gestellt) werden die Produktfunktionen einzelnen Teilprodukten zugeordnet. Die Teilprodukte werden gemäß ihrer Wichtigkeit für den Kunden angeordnet.

    ⇨ Eingabe für Detailplanung eines Softwareentwicklungsprozesses, siehe Kapitel 10.

    ⇨ hier hat man einen fließenden Übergang von der Analyse zum Design



Aufbau eines Pflichtenheftes - 8:

14. **Ergänzungen** (neuer Abschnitt):

    Hier wird alles aufgeführt, was sonst nicht ins Schema passt.

15. **Testfälle** (neuer Abschnitt):

    Hier werden alle Tests aus Anwendersicht aufgeführt, die bei der Abnahme des Software-Produkts durchlaufen müssen. Alle (wichtigen) Produktfunktionen und -Leistungen sollten durch entsprechende Testfälle abgedeckt werden.

16. **Glossar** (das Glossar aus dem Lastenheft wird fortgeschrieben)



5.7 Weitere Literatur

[Ba99] H. Balzert: *Lehrbuch der Objektmodellierung: Analyse und Entwurf*, Spektrum Akademischer Verlag (1999), 573 Seiten

Brauchbare Einführung in die Softwareentwicklung mit UML; insbesondere werden Themen wie die Gestaltung von Benutzeroberflächen und der Einsatz objektorientierter Datenbanken mit angesprochen.

[BC89] K. Beck, W. Cunningham: *A Laboratory For Teaching Object-Oriented Thinking*, in: Proc. OOPSLA'89, SIGPLAN Notices, Vol. 24, No. 10, ACM Press (1989), 1-6

Die Originalquelle zum Thema CRC-Karten (die für die Identifikation von Klassen benutzt werden).

[BD00] B. Bruegge, A.H. Dutoit: *Object-Oriented Software Engineering*, Prentice Hall (2000), 553 Seiten

Basiert auf den Erfahrungen mit der Durchführung von Praktika zur objektorientierten Softwareentwicklung an der TU München und der Carnegie Mellon University. Beschreibt eine ganze Reihe von Faustregeln für Projektmanagement, Anforderungsanalyse, Erstellung von UML-Diagrammen, … . Es ist schade, dass die verwendeten Beispiele dauernd wechseln.

[La98] Larman C.: *Applying UML and Patterns*, Prentice Hall (1998)

Eines der ersten UML-Bücher, das anhand eines durchgängigen Beispiels ein Vorgehensmodell zum Einsatz von UML vorstellt. Eine vereinfachte und abgeänderte Version dieses Vorgehensmodells wird in dieser Vorlesung benutzt.



6. Von der OO-Analyse zum Datenbank-Entwurf

Themen dieses Kapitels:

- kleiner Exkurs zu Datenbanken (Aufbau, Standards, Entwurfsprinzipien etc.)
- Klassendiagramme (Entity-Relationship-Diagramme) als Ausgangspunkt
- Datenbankschemabeschreibung (Datenmodellierung) mit SQL
- einfache Datenbankanfragen und Updates mit SQL

Achtung:

Die Entwicklung von Datenbanken ist ein komplexer Themenbereich, der üblicherweise in einer eigenen Vorlesung behandelt wird. Notgedrungen ist der Überblick hier über die Thematik sehr knapp und unvollständig!



Zur Erinnerung - Ablauf für Erstellung eines OO-Programms:





Planung

Kundenanforderungen
(z.B. als Lastenheft)
(siehe **Abschnitt 3.3**)

Anwendungsfälle
→ UML Use Cases
(siehe **Abschnitt 5.2**)

?

"Workflow"-Modell
→ UML Activity Diagrams
(siehe **Abschnitt 5.5**)

Analyse

konzept. Datenmodell
→ UML Class Diagrams
(siehe **Abschnitt 5.3**)

?

Datenbankschema
→ Definition mit SQL
(siehe **Abschnitt 6.3**)

DB-Anfragen/-Updates
→ mit SQL
(siehe **Abschnitt 6.5**)

Analyse → **Design**



Danksagung:

Dieses Kapitel ist ein Exzerpt einer Vorlesung und stützt sich im wesentlichen auf das sogenannte **Biberbuch** [SSH10] der Kollegen Saake, Sattler und Heuer ab.

Die Definition der Syntax von SQL-89 als EBNF wurde allerdings dem Skript zur Vorlesung **Datenbanksysteme** entnommen des Kollegens Janas von der Uni BW München.



6.1 Einführung in Datenbanken

"Eine Datenbank (auch Datenbanksystem) ist ein System zur Beschreibung, Speicherung und Wiedergewinnung umfangreicher Datenmengen, die von mehreren Anwendungsprogrammen oder Anwendern benutzt werden. Es besteht aus einer Datenbasis, in der die Daten abgelegt werden und dem Verwaltungsprogramm (Datenbasismanagement), das die Daten entsprechend der vorgegebenen Beschreibung abspeichern, auffinden oder weitere Operationen durchführen kann."

[Informatikduden]

"Eine Datenbank ist eine Sammlung von Informationen zu einem bestimmten Thema oder Zweck, wie z.B. dem Verfolgen von Bestellungen oder dem Verwalten einer Musiksammlung. Wenn Ihre Datenbank nicht oder nur teilweise in einem Computer gespeichert ist, müssen Sie die Informationen aus den verschiedenen Quellen selbst koordinieren und organisieren."

[MS-Access-Online-Hilfe]



Typische Beispiele für Datenbankanwendungen (Informationssysteme):

-
-
-
- Soziale Netze, Wissens-Datenbanken (Facebook, Google, ...)

Anfänge der Datenbanken (Speicherung dauerhafter Informationen):

- Anfang der 60er Jahre: elementare und anwendungsspezifische Dateien
- Ende der 60er Jahre: Dateiverwaltungssysteme (Sequential Access Memory, Index Sequential Access Memory) als Betriebssystem-Aufsätze
- 70er Jahre: erste "echte" Datenbanksysteme
- seitdem einerseits Standards, andererseits viele Speziallösungen für bestimmte Anwendungsbereiche (wie viele Terabytes Datenverwaltung bei Google, ...)



Probleme bei Datenverwaltung ohne Datenbanksysteme:



Die neun Codd'schen Anforderungen für Datenbanksysteme:

1. **Integration**:
2. **Operationen**:
3. **Katalog**:
4. **Benutzersichten**:
5. **Konsistenzüberwachung**:
6. **Datenschutz**:
7. **Transaktionen**:
8. **Synchronisation**:
9. **Datensicherung**:



Grundbegriffe der “Datenbankwelt”:

- **Datenbanksystem (DBS)**:

- **Dauerhafter Datenbestand**:

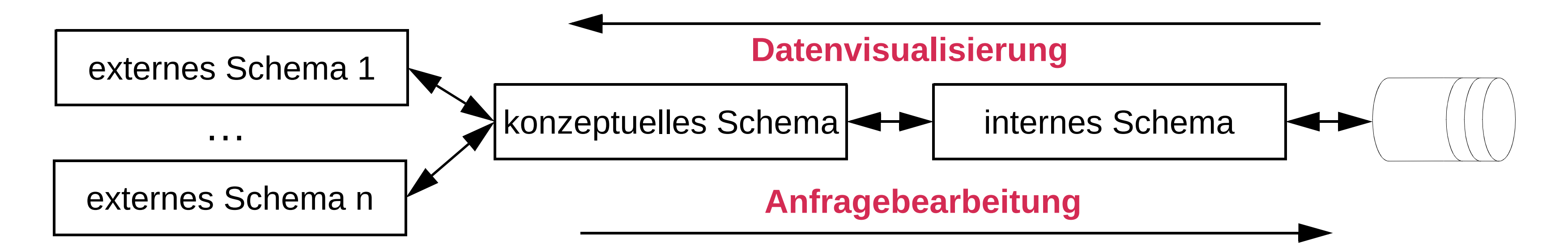
- **Datenbankschema**:

- **Datenbank (DB)**:

- **Datenbankmanagementsystem (DBMS)**:



Drei-Schichten-Schemaarchitektur (nach ANSI-SPARC):



Datenbankschema besteht aus

- einem internen Schema (Datenverwaltung auf Platte, Indexstrukturen, …)
- einem konzeptuellen Schema (Beschreibung der Gesamtstruktur, Integritätsbedingungen, …)
- i.a. mehreren externen Schemata (Sichten für verschiedene Anwendungen, Benutzergruppen, …)

Trennung Schema - Instanz

- Schema = Beschreibung der Daten (Metadaten)
- Instanz = Anwenderdaten (Datenbankzustand, -ausprägung)



Gründe für die Drei-Schichten-Schemaarchitektur:

- **physische Datenunabhängigkeit**: konzeptuelles Schema schützt Anwendungen vor Änderungen des internen Schemas
  - ⇨ Tuning-Maßnahmen am internen Schema leicht(er) möglich
  - ⇨ Anwendungen sind von Tuning-Maßnahmen nicht betroffen
  - ⇨ nur Abbildung konzeptuelles auf internes Schema ändert sich
- **logische Datenunabhängigkeit**: externe Schemata schützen Anwendungen vor Änderungen des konzeptionellen Schemas
  - ⇨ Umbenennungen, logische Reorganisationen, Erweiterungen einer Datenbank leicht(er) möglich
  - ⇨ logische DB-Reorganisationen für eine Gruppe von Anwendungen betreffen nicht andere Anwendungsgruppen
  - ⇨ nur Abbildung externes auf konzeptuelles Schema ändert sich



Entwicklungslinien = Historie der Datenbanksysteme:

ab 60er Jahre:

Datenbanksysteme basierend auf **hierarchischem Modell**, **Netzwerkmodell**:

- ⇨ Zeigerstrukturen zwischen Daten
- ⇨ schwache Trennung interne/konzeptuelle Ebene
- ⇨ navigierende Anfragesprachen (entlang Zeigerstrukturen)
- ⇨ Trennung Datenbanksprache/Programmiersprache

ab 70er Jahre:

relationale Datenbanksysteme:

- ⇨ Daten in Tabellenstruktur
- ⇨ 3-Ebenen-Konzept (externe, konzeptuelle, interne Ebene)
- ⇨ deklarative standardisierte Datenbanksprache (SQL)
- ⇨ Trennung Datenbanksprache/Programmiersprache



ab 80er Jahre:

Wissensbanksysteme (deduktive Datenbanken):

- ⇨ Daten in Tabellenstruktur
- ⇨ stark deklarative Anfragesprachen (logikbasiert)
- ⇨ integrierte Datenbankprogrammiersprache

objektorientierte (graphartige) Datenbanksysteme:

- ⇨ Daten in komplexeren Objektstrukturen
- ⇨ navigierende (oder deklarative) Anfragesprache
- ⇨ oft OO-Programmiersprache = Datenbankprogrammiersprache
- ⇨ oft keine vollständige Ebenentrennung

geographische Datenbanksysteme:

- ⇨ meist spezielle Erweiterungen relationaler oder objektorientierter Systeme
- ⇨ neben "normalen" Daten gibt es geometrische Daten (2D, 3D)
- ⇨ spezielle Indexstrukturen unterstützen „räumliche“ Anfragen



Relationale Datenbank(-Management)-Systeme (RDBMS):

Gemeinsame Merkmale:

- Drei-Ebenen-Architektur nach ANSI-SPARC
- standardisierte Datenbanksprache (SQL = Structured Query Language)
- Einbettung von SQL in kommerzielle Programmiersprachen
- Werkzeuge für (interaktive) Definition, Anfrage und Darstellung von Daten; Entwurf von Anwendungsprogrammen, Benutzeroberflächen
- kontrollierter Mehrbenutzerbetrieb, Datenschutz- und Datensicherheit

Beispiele echter RDBMS:

ORACLE, DB2, Ingres, Informix, …

Beispiele für “Pseudo”-RDBMS:

MS-Access, dBASE, …
(es fehlen u.a. Drei-Ebenen-Architektur, Optimierungen)



Einschub zur Definition von Menge und Tupel:

- Eine **Menge** ist eine Ansammlung (Kollektion) unterscheidbarer Dinge, die ihre **Elemente** genannt werden. Eine Menge enthält kein Element doppelt!

  Notation einer Menge:

  TITEL = { Datenbanken, Geoinformatik, Asterix der Gallier }

  Element (nicht) aus Menge:

  Datenbanken $\in$ TITEL, Dr. No. $\notin$ TITEL

  Vereinigung, Durchschnitt, Differenz von Mengen: $\cup$, $\cap$, \

- Ein **n-Tupel** ist eine (geordnete) Folge von genau n Elementen, seinen **Komponenten** (hier oft genannt: Spalten, Attribute).

  Beispiele für 4-Tupel = Quadrupel:

  (0001, Asterix der Gallier, 3-125… , Uderzo)
  (1201, Objektbanken, 3-111… , Heuer)
  (4711, Datenbanken, 3-765…, Vossen)



Einschub zur Definition von kartesischem Produkt und Relation, ... :

- Das **kartesische Produkt** zweier Mengen M und N ist die Menge aller Tupel, die als erste (zweite) Komponente ein Element aus M (N) enthalten:

  $M \times N = \{ (m, n) \mid m \in M, n \in N \}$

  Beispiel mit

  AUTOR = {Heuer, Saake, Uderzo, Goscini}, TITEL = {DB, Asterix}:

  AUTOR x TITEL =
  { (Heuer, DB), (Heuer, Asterix), (Saake, DB), (Saake, Asterix),
  (Uderzo, DB), (Uderzo, Asterix), (Goscini, DB), (Goscini, Asterix) }

- Eine **n-stellige Relation** über Mengen $M_1, \dots, M_n$ ist eine Teilmenge des kartesischen Produktes dieser Mengen.

  Beispiel für 2-stellige (binärer) Relation über AUTOR und TITEL:

  { (Heuer, DB), (Uderzo, Asterix) } ⊂ AUTOR x TITEL



Relationale Daten(-bank)-Definition mit Datenbankschema:

Schlüsselattribut

AUSLEIHE

INV_NR	NAME
4711	Meyer
1201	Schulz
0007	Müller
4712	Meyer

Relationenschema mit
- Spalten = Attribute
- Zeilen = Tupel

Relation = Instanz des Relationen-schemas

BUCH

INV_NR	TITEL	ISBN	AUTOR
0007	Asterix der Gallier	3-125…	Uderzo
1201	Objektbanken	3-111…	Heuer
4711	Datenbanken	3-765…	Vossen
4712	Datenbanken	3-891…	Ullman
4717	PASCAL	3-999…	Wirth

Attributnamen

Datenbank (DB) = Menge von Relationen = Tabellen
DB-Schema = Menge von Relationenschemata



Primär- und Fremdschlüssel von Relationen(-schemata):

- **Primärschlüssel** einer Relation (Relationenschemas): eine Menge von Attributen, die jede Zeile (Tupel) der Relation eindeutig identifiziert; es dürfen also keine zwei Zeilen einer Relationen die selben Attributwerte für alle Primärschlüsselattribute besitzen. Primärschlüssel können damit als Identifikatoren für alle Einträge in Datenbanktabellen genutzt werden.

- **Fremdschlüssel** einer Relation: Attribute einer Relation, die Primärschlüssel in einer anderen Relation sind. Die Fremdschlüsselwerte aller Zeilen einer Relation müssen als Primärschlüsselattribute von Zeilen der entsprechenden anderen Relation auftreten.

Damit können Primär- und Fremdschlüssel-Definitionen zur Beschreibung von Konsistenzbedingungen für Datenbanken eingesetzt werden!



Integritätsbedingungen = Konsistenzregeln zum Ausleih-Beispiel:

lokale Integritätsbedingungen:

- zu jedem Attribut in jedem Relationenschema gibt es eine Typdefinition, die zulässige Werte der entsprechenden Spalte festlegt (String, Int, …)
- Attribut INVENTARNR ist **(Primär-)Schlüssel** von BUCH
  - ⇨ in BUCH keine zwei Tupel mit demselben INVENTARNR-Wert
- Attribut INVENTARNR ist auch Primärschlüssel von AUSLEIHE
  - ⇨ in AUSLEIHE keine zwei Tupel mit demselben INVENTARNR-Wert

globale Integritätsbedingungen:

- Attribut INVENTARNR ist **Fremdschlüssel** von AUSLEIHE
  - ⇨ jeder INVENTARNR-Wert von AUSLEIHE muß auch in BUCH auftreten



Anfrageoperationen:

- **Selektion**: Zeilen (Tupel) auswählen

  SEL [NAME = ‘Meyer’](AUSLEIHE) ergibt:

INV_NR	NAME
4711	Meyer
4712	Meyer

- **Projektion**: Spalten (Attribute) auswählen:

  PROJ[INV_NR, TITEL](BUCH) ergibt:

INV_NR	TITEL
0001	Asterix der Gallier
1201	Objektbanken
4711	Datenbanken
4712	Datenbanken
4717	PASCAL



Anfrageoperationen - Fortsetzung:

- **Verbund (natural join)**: Tabellen verknüpfen über gleichbenannten Spalten und gleichen Werten:

  PROJ[INV_NR, TITEL](BUCH) JOIN SEL [NAME = ‘Meyer’](AUSLEIHE)

  ergibt:

INV_NR	TITEL	NAME
4711	Datenbanken	Meyer
4712	Datenbanken	Meyer

- **weitere Operationen**:
  - ⇨ Vereinigung, Durchschnitt, Differenz von gleich strukturierten Tabellen
  - ⇨ Umbenennung von Spalten (Attributen)

Achtung:



6.2 Operationen lassen sich beliebig kombinieren (schachteln) = **Relationenalge-**



braRelationaler Datenbankentwurf



Anforderungsanalyse:

siehe bisherige Kapitel der Vorlesung ...

Ergebnis:

- informale Beschreibung des Fachproblems (Texte, tabellarische Aufstellungen, Formblätter, Maskenentwürfe, …)
- Trennen der Informationen über Daten (Datenanalyse) von den Informationen über Funktionen (Funktionsanalyse)



Konzeptioneller Entwurf:

Erste formalere Beschreibung des Fachproblems (der zu speichernden Daten), heute in aller Regel mit **Klassendiagrammen**.

Vorgehensweise:

- **Modellierung** von Sichten (z.B. je Fachabteilung oder Benutzergruppe eine Sicht)
- **Analyse** der vorliegenden Sichten in Bezug auf
  - ⇨ Namenskonflikte (Homonyme: Kunde; Synonyme: Auto ↔ KFZ)
  - ⇨ Typkonflikte (verschiedene Strukturen für das gleiche Element)
  - ⇨ Wertebereichskonflikte (verschiedene Wertebereiche für ein Attribut)
  - ⇨ Bedingungskonflikte (z.B. verschiedene Schlüssel oder Kardinalitäten)
  - ⇨ Modellierungskonflikte (gleicher Sachverhalt unterschiedlich modelliert)
- **Integration** der Sichten in ein konzeptionelles Gesamtschema



Logischer Entwurf:

Umsetzung des konzeptionellen Entwurfs in Datenmodell des "Realisierungs"-DBMS, in aller Regel heute immer noch das **relationale Modell**.

Vorgehensweise:

1. (automatische) Transformation des konzeptionellen Schemas, also **Klassendiagramme (ER-Modelle) → relationale Modelle**

2. Verbesserung des relationalen Schemas anhand von Gütekriterien (Normalformen) durch entsprechende **Normalisierungsalgorithmen**

Es entsteht so ein logisches (relationales) Datenbankschema, das

- ⇨ Datenredundanzen weitgehend vermeidet
- ⇨ Konsistenzbedingungen aus dem ER-Modell weitgehend erhält



Verteilungsentwurf:

Sollen die Daten auf mehreren Rechnern verteilt vorliegen, muß Art und Weise der verteilten Speicherung festgelegt werden.

Verteilungsarten bei Relationen:

Kunde(KNr, Name, Adresse, PLZ, Kontostand)

- **horizontale Verteilung**:
  Kunde1 (KNr, Name, Adresse, PLZ, Kontostand) **where** PLZ < 50.000
  und
  Kunde2 (KNr, Name, Adresse, PLZ, Kontostand) **where** PLZ ≥ 50.000

- **vertikale Verteilung**:
  KundeAdr(KNr, Name, Adresse, PLZ)
  und
  KundeKonto(KNr, Kontostand)

(Verbindung kann über KNr-Attribut hergestellt werden)



Datendefinition:

Umsetzung des logischen Schemas in ein konkretes Datenbankschema mit

- ⇨ Data Definition Language (DDL)
- ⇨ Data Manipulation Language (DML)

eines konkreten DBMS (Oracle, MS-Access, …).

Vorgehensweise:

- ❑ Datenbankdeklaration mit DDL
- ❑ Realisierung der Integritätssicherung in …
- ❑ Definition der Benutzersichten
- ❑ [Definition von Update-Operationen, Anfragen, Auswertungen, …]



Physischer Entwurf:

"Tuning" der internen Abbildung der Relationen auf (Sekundär-)Speicher mit:

- Definition von Indexen (Indizes), die direkten (assoziativen) Zugriff auf alle Tupel einer Relation mit bestimmten Attributwerten erlauben
- Clusterverfahren zur Gruppierung von Daten im Sekundärspeicher, so dass zusammen benötigte Daten auf denselben Seiten liegen (vor allem bei OO-DBMS)
- …

Implementierung und Wartung:

Der ganze Rest … .



6.3 Vom Klassendiagramm zum Relationenmodell

Bei der Abbildung eines Klassendiagramms in das relationale Modell gehen wir grundsätzlich wie folgt vor:

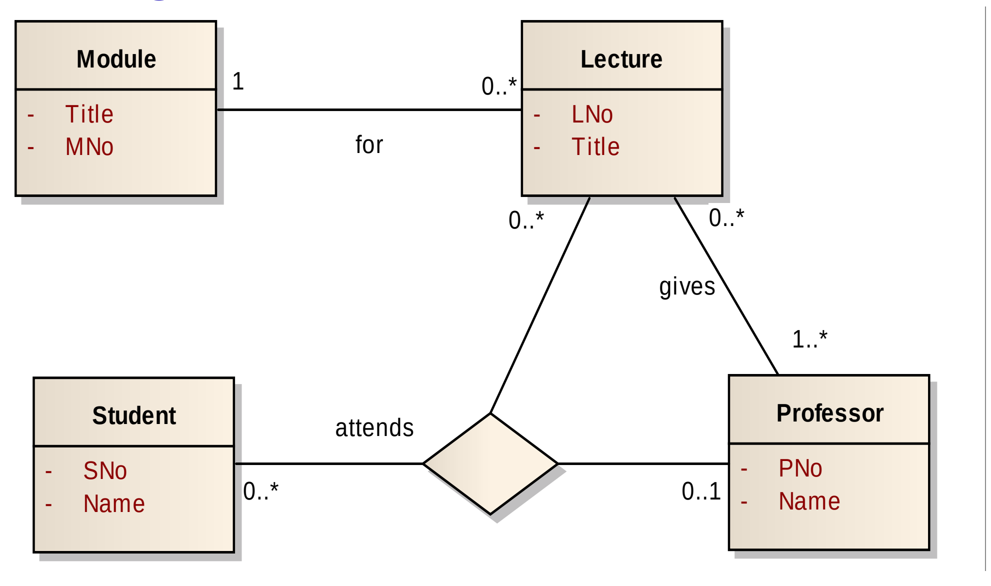
- Klassen und Assoziationen → i.a. jeweils ein eigenes Relationenschema
  - Klassen-Attribute → Attribute eines Relationschemas
  - Objekt-Identifikator-Attribute → (Primär-)Schlüssel v. Relationenschema
- Primärschlüsselwahl bei Relationship-Typen durch Multiplizitäten festgelegt
- erzeugte Relationenschemata können teilweise verschmolzen werden
- Einführung sogenannter „Fremdschlüsselbedingungen"
- besondere Behandlung von Vererbung notwendig (hier ausgeklammert)

Problem bei der Abbildung:

Korrekte Behandlung von Assoziationen = **Kapazitätserhaltung**.



Beispiel für Klassendiagramm in Relationenschema:



- ein Student hört (attends/at...) eine Menge von Vorlesungen
- jede Vorlesung hört er bei genau einem Professor (Professor zu einem konkreten Paar von Vorlesung und Student ist eindeutig festgelegt)
- jede Vorlesung gehört zu genau einem Modul (mit i.a. mehreren Vorlesungen)
- jede Vorlesung wird von ein bis mehreren Professoren gehalten (semesterweise abwechselnd)



Realisierung als Relationschemata:

- ⇨ garantiert, dass es keine zwei Studenten mit der selben Nummer gibt
- ⇨ garantiert, dass es keine zwei Professoren mit der selben Nummer gibt
- ⇨ garantiert, dass es keine zwei Module mit der selben Nummer gibt
- ⇨ garantiert, dass es keine zwei Vorlesungen mit der selben Nummer gibt
- ⇨ realisiert zudem die Assoziation for und garantiert damit,
  dass es zu einer Vorlesung immer genau ein Modul gibt

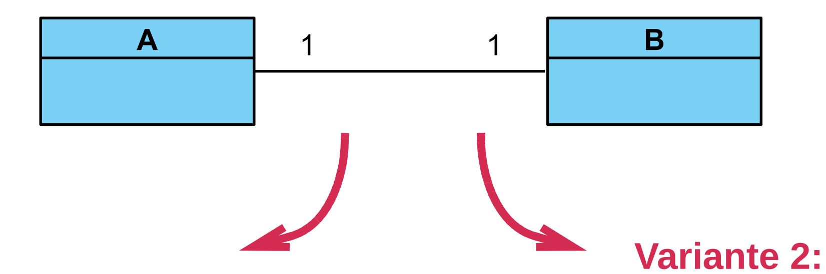


Realisierung als Relationschemata - Fortsetzung:

- ⇨ Relationenschema gives = {<u>LNo, PNo</u>}
  der aus LNo und PNo bestehende Primärschlüssel erlaubt beliebige Kombinationen von Vorlesungen und Professoren
- ⇨ nicht garantiert wird, dass es zu jeder Vorlesung mindestens einen Professor gibt (das geht auch über Primär- und Fremdschlüssel nicht)
- ⇨ es wird garantiert, dass es zu jeder Kombination aus Student und Vorlesung (die er tatsächlich hört) genau einen Professor gibt



Kapazitätserhöhende Abbildung von Assoziation auf eigene Relation:



Variante 1:

R = {A, B} -- Relationenschema
K = { {A} } -- R hat nur A als Schlüssel

Variante 2:

R = {A, B}
K = { {A}, {B} } -- A und B als Schlüssel (jeweils für sich)

Beispiele für Relationen:

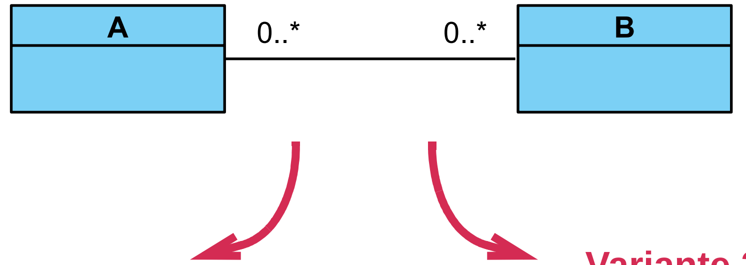
A	B
1	3
2	3

A	B
1	3
2	4

A	B
1	3
1	4



Kapazitätsvermindernde Abbildung von Assoziation auf eigene Relation:



Variante 1:

R = {A, B} -- Relationenschema
K = { {A} } -- R hat A als Schlüssel

Variante 2:

R = {A, B}
K = { {A, B} } -- Paar A, B als Schlüssel

Beispiele für Relationen:

A	B
1	3
2	3

A	B
1	3
2	4

A	B
1	3
1	4



6.4 Funktionale Abhängigkeiten und Normalformen

Aus Klassendiagrammen abgeleitete oder direkt entworfene Relationenschemata erzwingen oft **Redundanzen** bei der Speicherung von Daten.

⇨ Updates der entsprechenden Relationen führen zu Inkonsistenzen redundanter Daten (es wird nur an einer Stelle geändert, statt an allen Stellen).

Lösung des Problems:

- **Funktionale Abhängigkeiten** (1:n-Assoziationen im Klassendiagramm) liefern die Basis für die Erkennung von Redundanzen.
- **Normalformen** charakterisieren verschiedene Formen von Redundanzen auf der Basis von funktionalen Abhängigkeiten.
- **Transformationsverfahren** helfen bei der Elimination der Redundanzen.



Beispiel für Relation mit Redundanzen:

Bücher

ISBN	Titel	Autor	Auflage	Stichwort	Verlag
0-8053-1753-8	Princ. of DBS	Elmasri	1, 1989	RDB	Benj. Cummings
0-8053-1753-8	Princ. of DBS	Navathe	1, 1989	RDB	Benj. Cummings
0-8053-1753-8	Princ. of DBS	Elmasri	2, 1994	RDB	Benj. Cummings
0-8053-1753-8	Princ. of DBS	Navathe	2, 1994	RDB	Benj. Cummings
0-8053-1753-8	Princ. of DBS	Elmasri	1, 1989	Lehrbuch	Benj. Cummings
0-8053-1753-8	Princ. of DBS	Navathe	1, 1989	Lehrbuch	Benj. Cummings
0-8053-1753-8	Princ. of DBS	Elmasri	2, 1994	Lehrbuch	Benj. Cummings
0-8053-1753-8	Princ. of DBS	Navathe	2, 1994	Lehrbuch	Benj. Cummings

Beispiele für Änderungsanomalien:

- ⇨ **Update-Anomalie**: nur in einem Tupel “RDB” in “relationale DB” ändern
- ⇨ **Insert-Anomalie**: nur ein Tupel mit Stichwort “ER” in Relation einfügen
- ⇨ **Delete-Anomalie**: nur ein Tupel (egal welches) löschen



Entfernung der Redundanzen im Beispiel:

-
-
-
-

Was passiert, wenn man Auflage in Nr und Jahr zerlegt:

-

Was passiert, wenn man auch Adresse von Autoren speichern will:

-
-



Funktionale Abhängigkeiten (FD = Functional Dependency):

Eine **funktionale Abhängigkeit** $X \rightarrow Y$ für Attributmengen X und Y gilt für eine Relation r dann, wenn die Attributwerte für X die Attributwerte für Y festlegen:

Im Beispiel Bücher gilt:

- ⇨

Es gilt z.B. nicht:

- ⇨ ISBN → Autor
- ⇨ ISBN → Stichwort

Es gilt z.B. trivialerweise:

- ⇨
- ⇨

Schlüssel als Spezialfall einer funktionalen Abhängigkeit:

Eine Attributmenge $X \subseteq R$ ist **Schlüssel** des Relationenschemas *R*, wenn

- ⇨ für alle zulässigen Relationen *r(R)* die FD gilt: $X \rightarrow R$
- ⇨ *X* ist minimal(er Schlüssel) für *r*, also $\neg \exists X' \subset X : X' \rightarrow R$

Für die Definition “zulässiger Relationen” erweitert man das bisherige Relationen-schema R um einen Schlüssel K (oder eine Menge von Schlüsseln), also

$R' := (R, K)$ ist **erweitertes Relationenschema** mit Attributen *R* und $K \subset R$

Beispiele:

- ❑
- ❑



Relation zur Darstellung von Personen:

Personen

<u>PANr</u>	Vorname	Nachname	PLZ	Ort	Straße	HNr	Geb.datum
4711	Andreas	Heuer	18209	DBR	BHS	15	31.10.1958
5588	Gunter	Saake	39106	MD	STS	55	05.10.1960
0007	Andy	Schürr	82024	TK	AS	12	03.08.1961
7754	Andreas	Möller	18205	DBR	RS	31	25.02.1976
8832	Tamara	Jagellovsk	38106	BS	GS	12	11.11.1973
9912	Antje	Hellhof	18059	HRO	AES	21	04.04.1970
9999	Christa	Loeser	69121	HD	TS	38	10.05.1969

- Das „künstliche“ Attribut PANr wurde eingeführt, da es sonst keinen wirklich sinnvollen (Primär-)Schlüssel für die Relation gibt.
- Es gibt aber nicht nur die funktionale Abhängigkeit aller Attribute von PANr, sondern auch funktionale Abhängigkeiten zwischen den Bestandteilen der Adresse (siehe folgende Folien)!



Ziel des Datenbankentwurfs:

Das Ziel des relationalen Datenbankentwurfes ist es, Relationenschemata mit ihren (Primär-)Schlüsseln und Fremdschlüsseln so auszuwählen, dass

1. alle Anwendungsdaten aus den Basisrelationen hergeleitet werden können
2. nur semantisch sinnvolle/konsistente Anwendungsdaten dargestellt werden können
3. die Anwendungsdaten möglichst nicht-redundant dargestellt werden

Im Folgenden steht Forderung 3 im Vordergrund:

- die auf den nächsten Seiten definierten Normalformen charakterisieren **lokale Redundanzen** innerhalb einer Relation
- mit einer minimalen Anzahl von (normalisierten) Relationen vermeidet man **globale Redundanzen**



Die erste Normalform (1NF):

Relationenschemata in **erster Normalform** dürfen nur atomare Attribute besitzen.

Bemerkung:

Diese Forderung führt zu zusätzlichen Redundanzen, die durch die nachfolgenden Normalformen charakterisiert werden.

Beispiel:

InvNr	ISBN	Titel	Autoren
0007	0-8053-1753-8	Princ. of DBS	**{Elmasri, Navathe}**

wird zu

InvNr	ISBN	Titel	Autor
0007	0-8053-1753-8	Princ. of DBS	**Elmasri**
0007	0-8053-1753-8	Princ. of DBS	**Navathe**



Die zweite Normalform (2NF):

- Die **zweite Normalform** (2NF) fordert 1NF und verbietet zusätzlich partielle Abhängigkeiten zwischen einem Schlüssel und weiteren Nicht-Primattributen.
- Eine **partielle Abhängigkeit** liegt vor, wenn ein Attribut bereits von einer echten Teilmenge eines Schlüssels funktional abhängt.
- Ein **(Nicht-)Primattribut** ist ein Attribut, das (nicht) Bestandteil eines Schlüssels einer Relation ist (muß nicht Bestandteil des Primärschlüssels sein).

Beispiel:

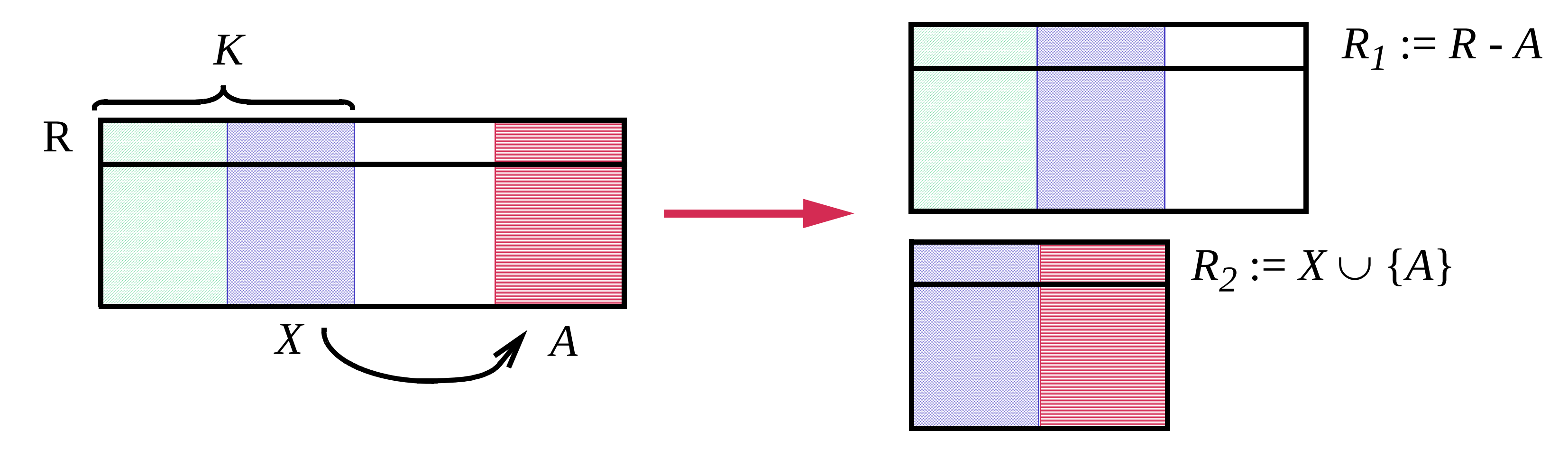
Mit {InvNr, Autor} als (Primär-)Schlüssel zu Beispiel auf voriger Seite gilt:

⇨ InvNr, Autor → InvNr, Titel, ISBN, Autor

⇨

Herstellung der zweiten Normalform:

Sei 2NF verletzt im Relationenschema *R* durch $X \rightarrow A$ mit Schlüssel *K*, $X \subset K$ sowie Nicht-Primattribut *A*. Dann eliminiert man die Verletzung durch



Beispiel:

Die dritte Normalform (3NF):

- Die **dritte Normalform** (3NF) fordert 1NF und verbietet zusätzlich transitive Abhängigkeiten zwischen einem Schlüssel und weiteren Nicht-Primattributen.



- Eine **transitive Abhängigkeit** liegt vor, wenn Attributmenge X funktional von Schlüssel K abhängt und Nicht-Primattributmenge Y funktional von X abhängt, also gilt: $K \rightarrow X$ und $X \rightarrow Y$ mit $K \neq X$
- Die 3NF beinhaltet immer die 2NF, da $X \subset K$ sein darf und dann $K \rightarrow X$ gilt; damit verbietet die 3NF auch $X \rightarrow Y$ für $X \subset K$ (Forderung der 2NF)

Beispiel:

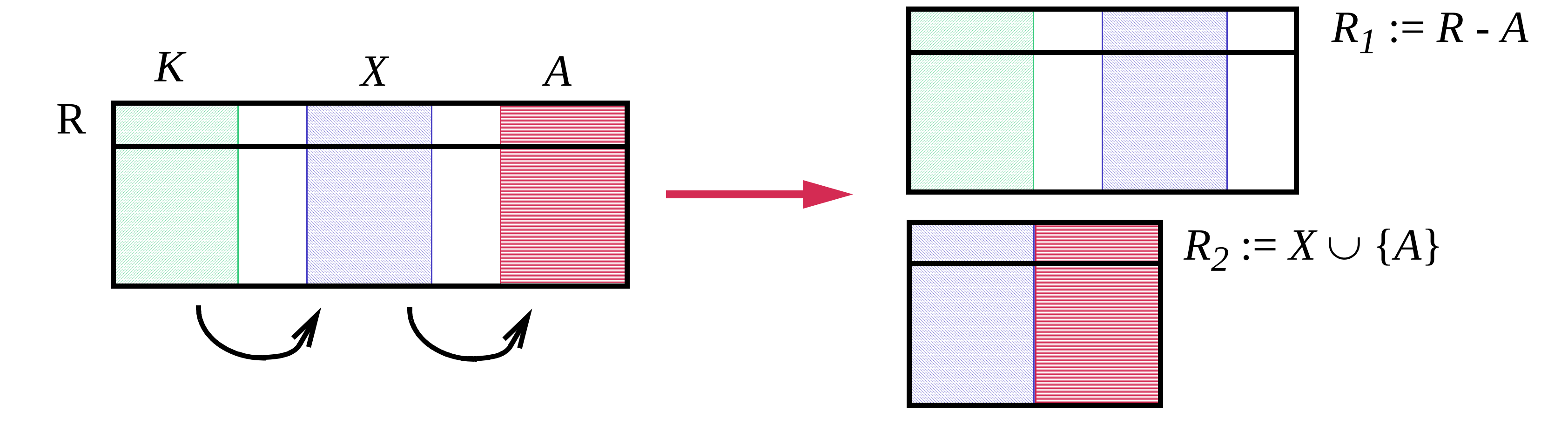
Mit {PANr} als (Primär-)Schlüssel für Beispiel auf Seite 291 gilt:

⇨

⇨

Herstellung der dritten Normalform:

Sei 3NF verletzt im Relationenschema *R* durch $K \rightarrow X \rightarrow A$ mit Schlüssel *K*, Attributmenge X und Nicht-Primattribut A. Dann eliminiert man die Verletzung durch



Beispiel:



Zusammenfassung der Schemaeigenschaften:

- **1NF**: nur atomare Attribute
- **2NF**: keine partielle Abhängigkeit eines Nicht-Primattributs von einem Schlüssel
- **3NF (S1)**: keine transitive Abhängigkeit eines Nicht-Primattributs von Schlüssel
- **...**
- **Minimalität (S2)**: minimale Anzahl von Relationen in …-NF

Bedeutung für die Praxis:

- Beim Datenbankentwurf beachtet man meist nur die Bedingungen (S1) und (S2), man entwirft also eine minimale Anzahl von Relationenschemata in 3NF.
- Noch „stärkere“ Normalformen als die 3NF sind für die Praxis oft zu einschränkend und werden deshalb eher selten verwendet.



6.5 Datenbankprogrammierung mit SQL

SQL wurde am IBM San Jose Research Laboratory entwickelt und im Laufe der Jahre auch SEQUEL oder RDL genannt. Die Entwicklung von Standard-SQL erfolgte in den folgenden Etappen:

- ⇨ bis Anfang der 80er Jahre Wildwuchs verschiedener Dialekte
- ⇨ erste Sprachstandardfestlegung um 1986
- ⇨ endgültiger Sprachstandard im Jahre 1989 = **SQL-89**
- ⇨ in Deutschland als DIN ISO 9075 im Jahre 1990 publiziert

Weitere wichtige Etappen:

- ⇨ im Jahre 1992 wird **SQL-92** = **SQL2** verabschiedet
- ⇨ seit Jahre 1999 gibt es **SQL:99** = **SQL3** = im wesentlichen heutiges **SQL**

In diesem Abschnitt wird Teilmenge von SQL-89 vorgestellt.



Syntax von SQL-DDL - Teil 1 (als EBNF):

```
<schema> ::=                CREATE SCHEMA <schema authorization clause> { <schema element> }
<schema authorization clause> ::= AUTHORIZATION <authorization identifier>

<schema element> ::=          <table definition>
                            | <privilege definition>            // wird nicht behandelt
                            | <view definition>                 // wird nicht behandelt

<table definition> ::=      CREATE TABLE <table name> '(' <table element> { ',' <table element> } ')'
<table element> ::=           <column definition>
                            | <table constraint definition>     // wird im Folgenden behandelt

<column definition> ::=     <column name> <data type> [ <default clause> ] { <column constraint> }
<data type> ::=             <character string type> | <exact numeric type> | …
<default clause> ::=        DEFAULT ( <literal> | USER | NULL )

<column constraint> ::=       NOT NULL [ <unique specification> ]
                            | <check constraint definition>     // wird nicht behandelt
                            | <references specification>        // wird im Folgenden behandelt
<unique specification> ::= UNIQUE | PRIMARY KEY
```



Erläuterungen zur SQL-Syntax - Teil 1:

- der authorization identifier legt den Eigentümer des deklarierten Schemas fest, der Zugriffsrechte (weiter-)vergeben kann, die privilege definition bestimmt die Zugriffsrechte im Detail
- mit einer view definition kann man Sichten auf den deklarierten Basistabellen einrichten, die das Ergebnis von Anfragen sind
- ein Schema umfaßt eine Menge von Tabellendefinitionen, die wiederum aus einer Menge von Spaltendefinitionen bestehen
- bei jeder Spalte kann neben dem Datentyp der Elemente ein Defaultwert (Initialwert) angegeben werden:
  - ⇨ eine Konstante (literal)
  - ⇨ der authorization identifier der Person, die das Tupel einträgt (USER)
  - ⇨ der Wert NULL für “definiert undefiniert”
- zu einzelnen Spalten können Integritätsbedingungen definiert werden



Beispiel für einfache Tabellendefinition:

```
CREATE TABLE Bücher(
        ISBN CHAR(10) NOT NULL PRIMARY KEY ,
        Titel CHAR(200) NOT NULL ,
        Verlagsname CHAR(30) DEFAULT 'Oldenbourg'  )
```

Probleme mit dieser Definition:

- wie legt man sinnvoll eine feste Anzahl von Zeichen für Buchtitel, Namen etc. fest (siehe Strings variabler Länge in SQL-92)
- wie definiert man einen aus mehreren Attributen zusammengesetzten Primärschlüssel (siehe nächste Folie)
- wie bringt man zum Ausdruck, daß zu einem Verlagsnamen in einer weiteren Tabelle zusätzliche Informationen abgelegt sind (siehe nächste Folie)



Syntax von SQL-DDL - Teil 2:

<table constraint definition> ::=
<unique constraint definition>
| <referential constraint definition>
| <check constraint definition> *// wird nicht behandelt*

<unique constraint definition> ::=
<unique specification> '(' <unique column list> ')'
<unique column list> ::= <<u>column name</u>> { ',' <<u>column name</u>> }

<referential constraint definition> ::=
FOREIGN KEY '(' <reference column list> ')' <references specification>

<references specification> ::=
REFERENCES <referenced table and columns>
<referenced table and columns> ::=
<<u>table name</u>> ['(' <reference column list> ')']

<reference column list> ::= <<u>column name</u>> { ',' <<u>column name</u>> }



Erläuterungen zur SQL-Syntax - Teil 2:

- es gibt drei Arten von **Integritätsbedingungen**:
  - unique constraints legen Primärschlüssel und Sekundärschlüssel fest; es handelt sich um **intrarelationale** Integritätsbedingungen auf einer Tabelle
  - referential constraints legen Fremdschlüsselbedingungen fest; es handelt sich um **interrelationale** Integritätsbedingungen zwischen zwei Tabellen
  - check constraints erlauben die Definition komplexer(er) Konsistenzbedingungen mit Hilfe von Anfragen; es handelt sich um **intrarelationale** Integritätsbedingungen
- jede Integritätsbedingung, die eine Spalte betrifft, kann bei der Definition dieser Spalte angegeben werden
- jede Integritätsbedingung über mehr als einer Spalte (z.B. zusammengesetzte Schlüssel) muß als separate Definition formuliert werden



Beispiel für Tabellendefinition mit Fremdschlüsselbedingung:

```
CREATE TABLE Bücher(

        ISBN CHAR(10) NOT NULL PRIMARY KEY,
        Titel CHAR(200),
        Verlagsname CHAR(30) NOT NULL
            REFERENCES Verlage(Verlagsname)   )
```

Beispiel für Tabellendefinition mit zusammengesetztem Primärschlüssel:

```
CREATE TABLE Buchexemplare(

        Nummer INT,
        ISBN CHAR(10),
        Ausleiher CHAR(30),

        PRIMARY KEY(Nummer, ISBN),
        FOREIGN KEY(ISBN) REFERENCES Bücher,
        FOREIGN KEY(Ausleiher) REFERENCES Benutzer(Name) )
```



Erweiterungen in SQL2 (SQL-92):

- Strings variabler Länge (aber feste Maximallänge)
- verschiedene Datentypen für Datums- und Zeitangaben
- neben Strings auch Bitfolgen fester/variabler Länge
- selbstdefinierte Datentypen (Aufzählungstypen etc.)
- mehr Schemaänderungen mit ADD, DROP und ALTER TABLE
- …

Erweiterungen in SQL3:

- objektorientierte Konzepte
- Tabellen in Tabellen
- …



SQL als Datenanfragesprache:

Der SQL-Anfrageteil beruht auf der Relationenalgebra und besteht im wesentlichen aus einem Konstrukt, dem sogenannten **SFW-Block** (für select … from … where):

select

bestimmt Projektionen auf Spalten mehrer Relationen (Tabellen)

from

legt zu verwendende (Basis-)Relationen fest

where

definiert Selektions- und Verbundbedingungen

group by

Auswertungen auf Untergruppen (Summenbildung, …)

having

Selektionsbedingungen für Bildung von Untergruppen



Syntax der select-Anweisung (SFW-Block):

<SFW block> ::= **SELECT** [**ALL** | **DISTINCT**] <select list> <table expression>

// Zuweisung mit INTO in Variablen einer Programmiersprache wird nicht behandelt

<select list> ::= * | <value expression> { , <value expression> }

<value expression> ::= … | <column specification> *// komplizierte Ausdrücke zugelassen*

<table expression> ::= <from clause>
[<where clause>]
[<group by clause>]
[<having clause>]

<from clause> ::= **FROM** <table reference> { , <table reference> }

<table reference> **::=** <<u>table name</u>> [<<u>variable name</u>>]

<where clause> ::= **WHERE** <search condition> *// wird nicht behandeltn*

<group by clause> ::= **GROUP BY** <column specification> { , <column specification> }

<column specification> ::= [<qualifier> .] <column name>

<qualifier> ::= <<u>table name</u>> | <<u>variable name</u>>

<having clause> ::= **HAVING** <search condition> *// wird nicht behandelt*



Auswertung einer select-Anweisung:

1. **from-Klausel**: im Prinzip wird hier das Kreuzprodukt aller angegebenen Tabellen gebildet (ohne Übereinanderlegen von Spalten bzw. Attributen)

2. **where-Klausel**: dann werden alle die Tupel der resultierenden (virtuellen) Tabelle selektiert, die die hinter WHERE angegebene Bedingung erfüllen

3. **group-by-Klausel**: es werden Tupelgruppen gebildet, deren angegebene Spalten-werte gleich sind

4. **having-Klausel**: schließlich werden aus den gebildeten Gruppen nur solche aus-gewählt, die die hinter HAVING angegebene Bedingung erfüllen

5. **select-Klausel**:

   ⇨ bildet aus der berechneten Tabelle mit Gruppen eine neue Tabelle, die je Gruppe ein Tupel enthält (Verdichtung mit Aggregationsfunktionen)

   ⇨ selektiert alle oder einzelne Spalten und berechnet ggf. vollständig neue Spaltenwerte aus den Spaltenwerten jeweils eines betrachteten Tupels



Beispiel mit Büchern wieder aufgegriffen:

Autoren

ISBN	Autor
0-8053-1753-8	Elmasri
0-8053-1753-8	Navathe
3-8244-2021-X	Schürr
3-8244-2075-9	Zündorf

Ausleihe

ISBN	Name
0-8053-1753-8	Schmitz
0-8053-1753-8	Derichsweiler
3-8244-2021-X	Radermacher
3-8244-2075-9	Radermacher

Bücher

ISBN	Titel	Verlag
0-8053-1753-8	Principles of Database Systems	Benj. Cummings
3-8244-2021-X	Operationales Spezifizieren mit …	Deutscher Universitäts-Verlag
3-8244-2075-9	PROgrammierte GRaphERsetzungsSysteme	Deutscher Universitäts-Verlag



Einfachste SQL-Anfrage:

SELECT * FROM Autoren, Ausleihe

⇨ bildet kartesisches Produkt der beiden Tabellen (alle möglichen Kombinationen aller Zeilen beider Tabellen) und selektiert mit * alle Spalten der resultierenden Tabelle

Autoren.ISBN	Autor	Ausleihe.ISBN	Name
0-8053-1753-8	Elmasri	0-8053-1753-8	Schmitz
0-8053-1753-8	Navathe	0-8053-1753-8	Schmitz
3-8244-2021-X	Schürr	0-8053-1753-8	Schmitz
3-8244-2075-9	Zündorf	0-8053-1753-8	Schmitz
0-8053-1753-8	Elmasri	0-8053-1753-8	Derichsweiler
0-8053-1753-8	Navathe	0-8053-1753-8	Derichsweiler
3-8244-2021-X	Schürr	0-8053-1753-8	Derichsweiler
3-8244-2075-9	Zündorf	0-8053-1753-8	Derichsweiler
0-8053-1753-8	Elmasri	3-8244-2021-X	Radermacher
…	…	…	…



SQL-Anfrage mit Verbundbildung:

```
SELECT Autoren.ISBN, Autor, Name
    FROM Autoren, Ausleihe
    WHERE Autoren.ISBN = Ausleihe.ISBN
```

- ⇨ berechnet den Verbund der beiden Tabellen über ISBN (natural join)
- ⇨ anstelle von “=” geht auch Vergleich auf “<>” oder “<“ oder …

Autoren.ISBN	Autor	Name
0-8053-1753-8	Elmasri	Schmitz
0-8053-1753-8	Navathe	Schmitz
0-8053-1753-8	Elmasri	Derichsweiler
0-8053-1753-8	Navathe	Derichsweiler
3-8244-2021-X	Schürr	Radermacher
3-8244-2075-9	Zündorf	Radermacher



Syntax der SQL-Suchbedingungen (Auszug):

<search condition> ::= <boolean term> | <search condition> **OR** <search condition>
<boolean term> ::= <boolean factor> | <boolean term> **AND** <boolean term>
<boolean factor> ::= [**NOT**] <boolean primary>
<boolean primary> ::= <predicate> | '(' <search condition> ')'

<predicate> ::= <comparison predicate> | <between predicate> | <in predicate>
| <like predicate> | <null predicate> | <quantified predicate> | <exists predicate>

<comparison predicate> ::= <value expression> <comp op> (<value expression> | <subquery>)
<comp op> ::= = | <> | < | > | <= | >=

<subquery> ::= '(' SELECT [**ALL** | **DISTINCT**] <result specification> <table expression> ')'
<result specification> ::= * | <value expression>

<between predicate> ::=
<value expression> **BETWEEN** <value expression> **AND** <value expression>

<in predicate> ::= <value expression> [**NOT**] **IN** (<subquery> | <in value list>)
<in value list> ::= <value specification> { , <value specification> }

…



Erläuterungen zu der Syntax von Suchbedingungen:

- es lassen sich über den Spalten/Attributen einer Tabelle beliebige **aussagenlogische Formeln** aufbauen
- man beachte, dass ein solches Prädikat immer bezogen auf **ein Tupel** der betrachteten Tabelle ausgewertet wird
- bestimmte Prädikate erlauben das **Schachteln** von Anfragen
- das **between**-Prädikat ist eine Abkürzung für zwei Vergleiche mit <= und >=
- das **in**-Prädikat überprüft, ob ein bestimmter Wert in einer Menge von Werten enthalten ist (die geschachtelte Anfrage muß Tabelle mit einer Spalte berechnen)
- mit dem **like**-Prädikat kann man reguläre Ausdrücke in Zeichenketten suchen
- mit dem **null**-Prädikat überprüft man, ob eine bestimmte Spalte eines Tupels einen Wert (un-)gleich NULL hat
- mit dem **quantified**-Prädikat überprüft man, ob alle/einige Einträge einer Spalte der Tabelle gleich/ungleich/… einem bestimmten Wert sind
- das **exist**-Prädikat überprüft, ob eine Teilanfrage eine leere Tabelle liefert



Einige Beispiele für SQL-Prädikate:

```
SELECT ISBN FROM Bücher
        WHERE Preis BETWEEN 20.0 AND 100.0
```

⇨

```
SELECT * FROM Bücher
        WHERE Verlagsname LIKE ’Ben%Cummings’
```

⇨

```
SELECT Titel FROM Bücher
        WHERE ’Radermacher’ IN ( SELECT Name FROM Ausleihe
                                         WHERE ISBN = Bücher.ISBN )
```

⇨



SQL als Datenmanipulationssprache

Für die Manipulation von Tabellen (Relationen) gibt es in SQL

- die **insert**-Anweisung für das Einfügen eines einzelnen Tupels oder einer Menge von Tupeln; die einzufügenden Tupel mit ihren Werten werden
  - ⇨ entweder explizit angegeben
  - ⇨ oder durch eine Query bestimmt
- die **delete**-Anweisung für das Löschen eines einzelnen Tupels oder einer Menge von Tupeln; die zu löschenden Tupel werden
  - ⇨ durch eine Query bestimmt
- die **update**-Anweisung für das Verändern einzelner Attribute eines einzelnen Tupels oder einer Menge von Tupeln; die zu verändernden Tupel werden
  - ⇨ durch eine Query bestimmt



Syntax der SQL-Änderungsanweisungen:

<insert statement> ::= **INSERT INTO** <table name> ['(' <insert column list> ')']
(**VALUES** '(' <insert value list> ')' | <query specification>)
<insert column list> ::= <column name> { , <column name> }
<insert value list> ::= <insert value> { , <insert value> }
<insert value> ::= <value specification> **| NULL**

<update statement> ::= **UPDATE** <table name> **SET** <set clause list>
[**WHERE** <search condition>]
<set clause list> ::= <set clause> { , <set clause> }
<set clause> ::= <column name> = (<value expression> **| NULL**)

<delete statement> ::= **DELETE FROM** <table name>
[**WHERE** <search condition>]

Achtung: Neben den hier vorgestellten update- und delete-Anweisungen gibt es noch andere, die an einer bestimmten Stelle in einer Tabelle ändern oder löschen. Diese Anweisungen werden bei der Einbettung von SQL-Anweisungen in Programme benutzt.



Beispiele für SQL-Einfügeoperationen:

```
INSERT INTO Bücher
    VALUES ('3-486-25053-1', 'Datenbanksysteme', 'Oldenbourg', )
```

die Anweisung fügt in die Tabelle Bücher eine neue Zeile mit vollständiger Angabe aller Spaltenwerte ein

```
INSERT INTO Bücher (ISBN, Titel)
    VALUES ('3-486-25053-1', 'Datenbanksysteme)
```

die Anweisung

- ⇨ fügt in die Tabelle Bücher eine neue Zeile ein
- ⇨ setzt bei den Spalten ISBN und Titel die angegebenen Werte ein
- ⇨ und setzt für die verbleibende Spalte den Defaultwert (oder NULL) ein

```
INSERT INTO Buchexemplare (ISBN, Nummer)
    ( SELECT ISBN, 1 FROM Bücher )
```

die Anweisung erzeugt für jedes bekannte Buch ein Exemplar mit Nummer 1



Beispiele für Delete- und Update-Anweisungen:

```
DELETE FROM Ausleihe WHERE Name = 'Bond'
```

die Anweisung löscht alle Ausleihe-Einträge der Person mit Namen 'Bond'

```
UPDATE Ausleihe SET Name = 'Rövenich'
        WHERE Name = 'Weigmann'
```

die Anweisung ändert den Namen einer Person in der gesamten Tabelle

```
UPDATE Bücher SET Preis = Preis + (Preis / 100 * 2)
        WHERE ISBN LIKE '3%'
```



6.6 Transaktionen und 2-Phasen-Protokoll

Transaktion:

Eine Transaktion ist ein Block von Lese- und Änderungsoperation, die eine "semantisch" abgeschlossene Operation auf einer Datenbank durchführen. Sie besitzt folgende Eigenschaften:

- ⇨ **Atomicity**: alle Änderungsoperationen werden ausgeführt oder gar keine
- ⇨ **Consistency**: nach Ende der Transaktion ist Datenbank in konsistentem Zustand (Überprüfung von Integritätsbedingungen)
- ⇨ **Isolation**: parallel laufende Transaktionen "stören" sich nicht gegenseitig (sondern werden aus Anwendersicht hintereinander ausgeführt)
- ⇨ **Durability**: Wirkung einer einmal erfolgreich beendeten Transaktion ist dauerhaft und wird nicht einmal durch technische Störungen rückgesetzt

Man spricht von den **ACID**-Eigenschaften einer Transaktion.



Beispiele für Notwendigkeit von Transaktionen:

- **Atomicity**: Was, wenn DBMS bei einer Überweisung nach Erniedrigung des Kontostands von K1 und vor Erhöhung des Kontostands von K2 abstürzt?
  - ⇨ **No**:
- **Consistency**: Was passiert, wenn nach einer Überweisung von Konto K1 auf Konto K2 Dispositionskredit für Konto K1 (weit) überzogen ist.
  - ⇨ **No**:
- **Isolation**: Was passiert wenn "gleichzeitig" zwei Bankfilialen den Stand eines Kontos lesen und jeweils um einen bestimmten Betrag erniedrigen wollen?
  - ⇨ **No**:
- **Durability**: Was passiert, wenn nach der Durchführung einer Reihe von Überweisungen von einer Bank auf eine andere das DBMS der einen Bank abstürzt?
  - ⇨ **No**:



Sperrkonzept für serialisierbare Schedules:

Durch Sperren muß vermieden werden, daß von T1 gelesene Attributwerte und noch in Bearbeitung befindliche Attributwerte von T2 parallel verändert werden.

Sperrgranularität:

- ⇨ ganze Tabelle (für umfangreichere Operationen, Auswertungen)
- ⇨ einzelne Zeile (bei selektivem Lesen oder Ändern)
- ⇨ einzelnes Attribut (bei “sehr” selektivem Lesen oder Ändern)

Übliche Sperrarten:

- ⇨ Lesesperre erlaubt weitere Lesesperre, aber keine Schreibsperre
- ⇨ Schreibsperre erlaubt keine weitere Sperre

Übliche Sperrprotokolle:

- ⇨ Zwei-Phasen-Protokoll: alle Sperranforderungen vor allen Sperrfreigaben
- ⇨ striktes Zwei-Phasen-Protokoll: Sperrfreigaben mit Transaktionsende



6.7 Ausblick

Die relationalen Datenbankmanagementsysteme (RDBMS) dominieren weiterhin den Markt. So genannte „**Nicht-Standard-DBMS**“ bzw. „**NoSQL-Ansätze**“ sind auf ganz spezielle Anwendungsbereiche beschränkt und besitzen meist nur eine kleine Teilmenge der Funktionalität der RDMBS.

Ein wichtiger (neuer?) Bereich sind die so genannten „Key-Value“-Datenspeicher für

- Speicherung von vielen Terabytes an strukturierten Daten
- hochgradig verteilte / redundante / ausfallsichere Datenhaltung
- schwache Konsistenzgarantieren (eventual consistency; Konsistenz von Replikaten wird nicht sofort, sondern nur irgendwann garantiert)
- einfache Schnittstellen zu Anwendungen

Beispiel: Google’s verteiltes Datenhaltungssystem **BigTable**.



6.8 Weitere Literatur

[EN94] R. Elmasri, S.B. Navathe: Grundlagen von Datenbanksystemen (*Fundamentals of Database Systems)*; Pearson Studium, 3. Auflage (2009), 535 Seiten
(Original in Englisch bei Benjamin/Cummings Publ. Company publiziert)

[SSH10] G. Saake, K.-U. Sattler, A. Heuer, : *Datenbanken - Konzepte und Sprachen*; MITP-Verlag, 4. Auflage (2010); 780 Seiten



7. Von der OO-Analyse zum Software-Entwurf

Themen dieses Kapitels:

- Feinheiten der Datenmodellierung mit UML-Klassendiagrammen
- Modellierung von Abläufen mit Interaktionsdiagrammen
- Beschreibung reaktiven Systemverhaltens mit Automaten
- Brücke von „reinen“ Analyse- zu „reinen“ Design-Tätigkeiten

Achtung:

Die hier vorgestellten Diagrammelemente und Techniken können also sowohl zur Erstellung sehr präziser (ausführbarer) Anforderungsdokumente (Pflichtenhefte) als auch für das Design von Software eingesetzt werden.



Zur Erinnerung:

Planung

Kundenanforderungen
(z.B. als Lastenheft)
(siehe **Abschnitt 3.3**)

Anwendungsfälle
→ UML Use Cases
(siehe **Abschnitt 5.2**)

?

"Workflow"-Modell
→ UML Activity Diagrams
(siehe **Abschnitt 5.5**)

Analyse

konzept. Datenmodell
→ UML Class Diagrams
(siehe **Abschnitt 5.3**)

?

feines Verhaltensmodell
→ UML Interaction Diag.
(siehe **Abschnitt 7.3**)

feines Klassenmodell
→ Class Diagram mit Ops.
(siehe **Abschnitt 7.1**)

?

reaktives Verhalten
→ UML State Machines
(siehe **Abschnitt 7.4**)

Analyse (→ Design)



7.1 Verfeinerte Klassendiagramme

Aufgabe:

Verfeinertes Klassendiagramm erzeugen, das

- ⇨ **Operationen** für Zugriff auf Objekteigenschaften anbietet
- ⇨ **Zusammenhänge** deklarierter Klassen präzisiert, …
- ⇨ **Vererbungshierarchien** exakter beschreibt
- ⇨ zusätzliche **Konsistenzbedingungen** präzise festlegt

Vorgehensweise:

- ⇨ Einzelaktivitäten der textuellen Anwendungsfallbeschreibungen oder in Aktivitätsdiagrammen in Klassenoperationen umwandeln
- ⇨ Assoziationen und Vererbungsbeziehungen zwischen Klassen genauer beschreiben (und ergänzen)
- ⇨ Konsistenzbedingungen in Umgangssprache oder Prädikatenlogik (Object Constraint Language) definieren



Identifikation von Operationen:

Operationen →

ReservationContract
contractId : Number {readonly} category : CatType deposit : Currency = 500 period : DateRange <u>noOfContracts : int = 0</u>
<u>startContractCreation(id:Number)</u> setCategory(category:CatType) getCategory():CatType ... setVehicleData(...) getVehicleData(...) cancelContractCreation() finishContractCreation() cancelContract() removeFulfilledContract()

oder eine Operation CreateContract mit allen notwendigen Daten

Operationen werden für Objekterzeugung und -löschung sowie für Zugriff auf Attribute, assoziierte Objekte, Zustandsänderungen, … eingeführt.



Aufbau von Operationen:

<Sichtbarkeit> <Name> <Parameterliste> : <Rückgabetyp>

- **Sichtbarkeit**: wie bei Attributen
- **Name**: ein Name, der innerhalb der Klasse (zusammen mit Param.) eindeutig ist
- **Parameterliste** mit Parametern der folgenden Form:

  <Richtung> <Name> : <Typ> = <Defaultwert>

  Richtung ∈ { in, out, inout } wie in Ada mit „in“ als Default-Wert und
  - ⇨ **in**: Eingabeparameter für Operation
  - ⇨ **out**: Ausgabeparameter für Operation (von Java nicht unterstützt)
  - ⇨ **inout**: Ein- und Ausgabeparameter für Operation
- **Rückgabetyp**: Typ des Ergebniswerts der Operation (Funktion);
  nicht jede Operation (ist eine Funktion und) muss Rückgabewert besitzen
- **Klassenoperationen** (in Java static genannt) werden unterstrichen



Faustregeln für die Festlegung von Operationen:

- Operationen für lesenden/schreibenden **Zugriff auf Attribute** dürfen fehlen (lassen sich später automatisch generieren)
- **Konstruktoren** dürfen ebenfalls fehlen, falls es sich nur um parameterlose Standard-Konstruktoren handelt (lassen sich auch automatisch generieren)
- Operationen für die Manipulation von **Assoziationen** können ebenfalls fehlen (lassen sich auch automatisch generieren)
- **Aktivitäten** aus Anwendungsfallbeschreibungen/Aktivitätsdiagrammen, die auf Instanzen genau einer Klasse operieren, werden dieser Klasse zugeordnet
- andere Aktivitäten werden
  - ⇨ in Teilaktivitäten zerschlagen oder
  - ⇨ bei “umfassenden” Objekten deklariert (siehe Objektkomposition) oder
  - ⇨ eigenen Transaktionsobjekten zugeordnet



Übersetzung in Java-Code:

```
public class ReservationContract {
    public static ReservationContract StartContractCreation(Number id) {
        ReservationContract reservationContract = new ReservationContract();
        reservationContract.contractId = id;
        return reservationContract;
    }
    public DateRange getPeriod(){ return period; }
    public void setPeriod(DateRange period){ this.period = period; }
        …
    public void cancelContractCreation() { … ; /* Verweise auf Contract löschen */ }
    public void FinishContractCreation() { noOfContracts++; }
    public void CancelContract() {
        noOfContracts--; … ; /* Verweise auf Contract löschen */
    }
    private ReservationContract {}; /* der Konstruktor der Klasse */
    private Number contractId; /* readonly */
    private DateRange period;
    private CatType category;
    private Currency deposit = 500;
    private static int noOfContracts = 0;
}
```



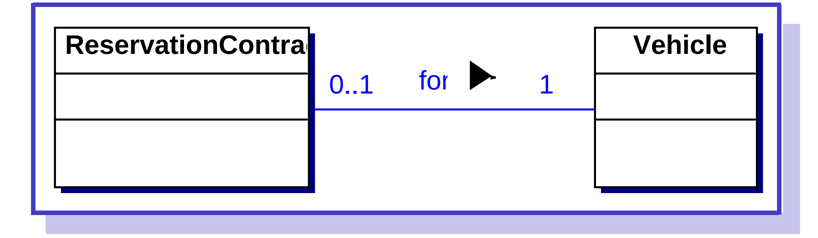
Von CASE-Tool generierter Java-Code:

```
public class ReservationContract {
  public static ReservationContract StartContractCreation(Number id) {
    ReservationContract reservationContract = new ReservationContract();
    reservationContract.contractId = id;
    return reservationContract;
  }
  public DateRange getPeriod(){ return period; }
  public void setPeriod(DateRange period){ this.period = period; }
    …
  public void cancelContractCreation() { … ; /* Verweise auf Contract löschen */ }
  public void FinishContractCreation() { noOfContracts++; }
  public void CancelContract() {
    noOfContracts--; … ; /* Verweise auf Contract löschen */
  }
  private ReservationContract {}; /* der Konstruktor der Klasse */
  private Number contractId; /* addonly */
  private DateRange period;
  private CatType category;
  private Currency deposit = 500;
  private static int noOfContracts = 0;
}
```



Unvollständiger naiver generierter Java-Code für Assoziationen:

Client der Assoziation



Supplier der Assoziation

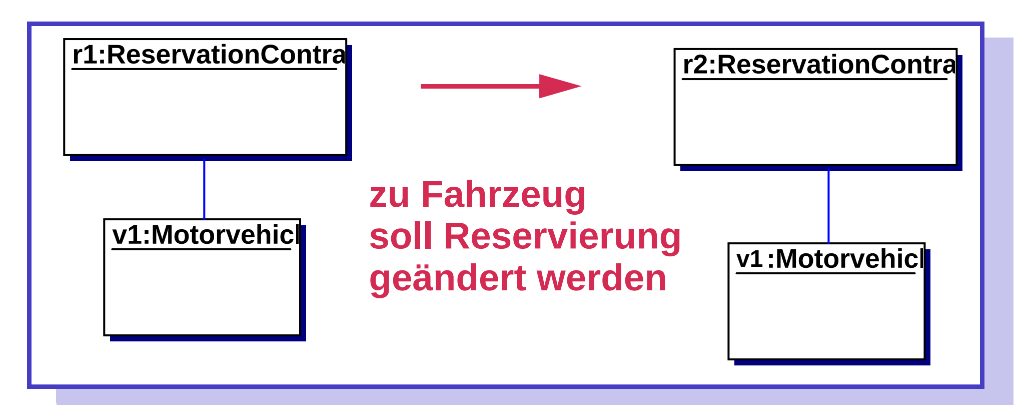
```
public class ReservationContract {
   public Vehicle getLnkVehicle(){ return lnkVehicle; }
   public void setLnkVehicle(Vehicle lnkVehicle){ this.lnkVehicle = lnkVehicle; }
   private Vehicle lnkVehicle;   /* Vorwärts-Link auf Supplier = Fahrzeug */
}
public class Vehicle {
   public ReservationContract getLnkRevReservationContract(){ return lnkrevReservationContract; }
   public void setLnkReservationContract(ReservationContract lnkrevReservationContract){
      this.lnkrevReservationContract = lnkrevReservationContract;
   }
   private ReservationContract lnkrevReservationContract;   /* Rückwärts-Link auf Client = Vertrag */
}
```



Kritik an dieser Umsetzung in Code:



Warum ist der generierte Code naiv?



- Änderung von v1.lnkrevReservationContract von r1 auf r2 mit der Methode setLnkReservationContract zieht nicht automatisch
  - ⇨ Änderung von r1.lnkMotorvehicle auf null
  - ⇨ Änderung von r2.lnkMotorvehicle von v2 auf v1
  - ⇨ Änderung von v2.RentalContract auf null

  nach sich (oder umgekehrt)
- **Im übrigen**: Beispiel ist nicht sinnvoll, da es zu einem Motorvehicle mehr als einen RentalContract geben wird.



Java-Code für Assoziation mit Konsistenzbewahrung - 1:

```
class ReservationContract{  …
   public Motorvehicle getMotorvehicle(){
      return lnkMotorvehicle;
   }
   public void setMotorvehicle(Motorvehicle newMotorvehicle) {
      if (this.lnkMotorvehicle != newMotorvehicle) {
         if (this.lnkMotorvehicle != null) {
            Motorvehicle oldMotorvehicle = this.lnkMotorvehicle ;
            this.lnkMotorvehicle = null;
            oldMotorvehicle.setReservationContract(null);
         };
         if (newMotorvehicle != null) {
            this.lnkMotorvehicle = newMotorvehicle;
            newMotorvehicle.setReservationContract(this);
         }
      }
   }
```

Achtung: die Methode MotorVehicle.setReservationContract und die Methode ReservationContract.setMotorVehicle rufen sich gegenseitig auf



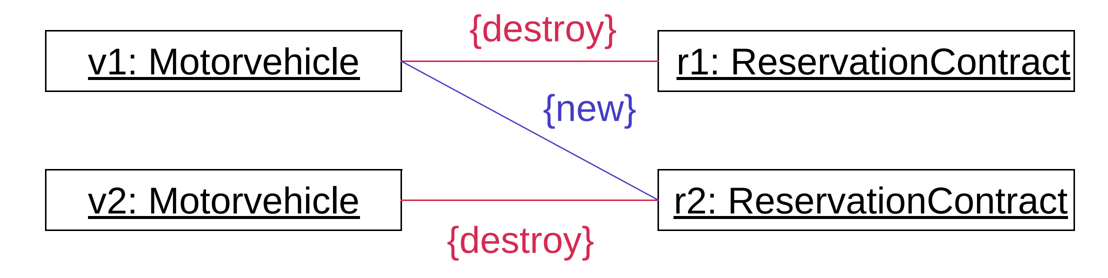
Java-Code für Assoziation mit Konsistenzbewahrung - 2:

```
class Motorvehicle{  …
   public ReservationContract getReservationContract(){
      return lnkrevReservationContract;
   }
   public void setReservationContract(ReservationContract newReservationContract) {
      if (this.lnkrevReservationContract != newReservationContract) {
         if (this.lnkrevReservationContract != null) {
            ReservationContract oldReservationContract = this.lnkrevReservationContract;
            this.lnkrevReservationContract = null;
            oldReservationContract.setMotorvehicle(null);
         };
         if (newReservationContract != null) {
            this.lnkrevReservationContract = newReservationContract;
            newReservationContract.setMotorvehicle(this);
         }
      }
   }
```

Achtung: die Methode MotorVehicle.setReservationContract und die Methode ReservationContract.setMotorVehicle rufen sich gegenseitig auf



Rekursive Aufrufe am Beispiel:



v1.setReservationContract(r2):

1) r1.setMotorvehicle(null) nach v1.lnkrevReservationContract = null:

2) ⊢ v1.setReservationContract(null) nach r1.lnkMotorvehicle = null:
   Aufrufkette terminiert, da bereits v1.lnkrevReservationContract = null

3) r2.setMotorVehicle(v1) nach v1.lnkrevReservationContract = r2:

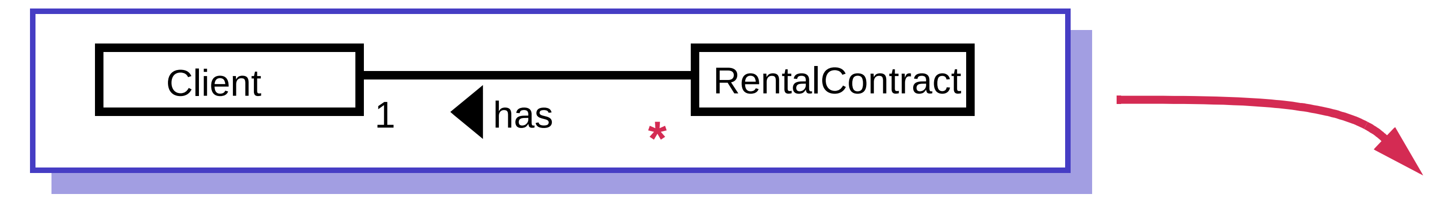
4) ⊢ v2.setReservationContract(null) nach r2.lnkMotorvehicle = null:

5) ⊢ r2.setMotorVehicle(null) nach v2.lnkrevReservationContract = null:
   Aufrufkette terminiert, da bereits r2.lnkMotorvehicle = null

6) ⊢ v1.setReservationContract(r2) nach r2.lnkMotorvehicle = v1:
   Aufrufkette terminiert, da bereits v1.lnkrevReservationContract = r2



Codeerzeugung für Assoziationen mit höherer Kardinalität:




```
class Client {
...
private RentalContract has_RentalContract[];
...
}
```

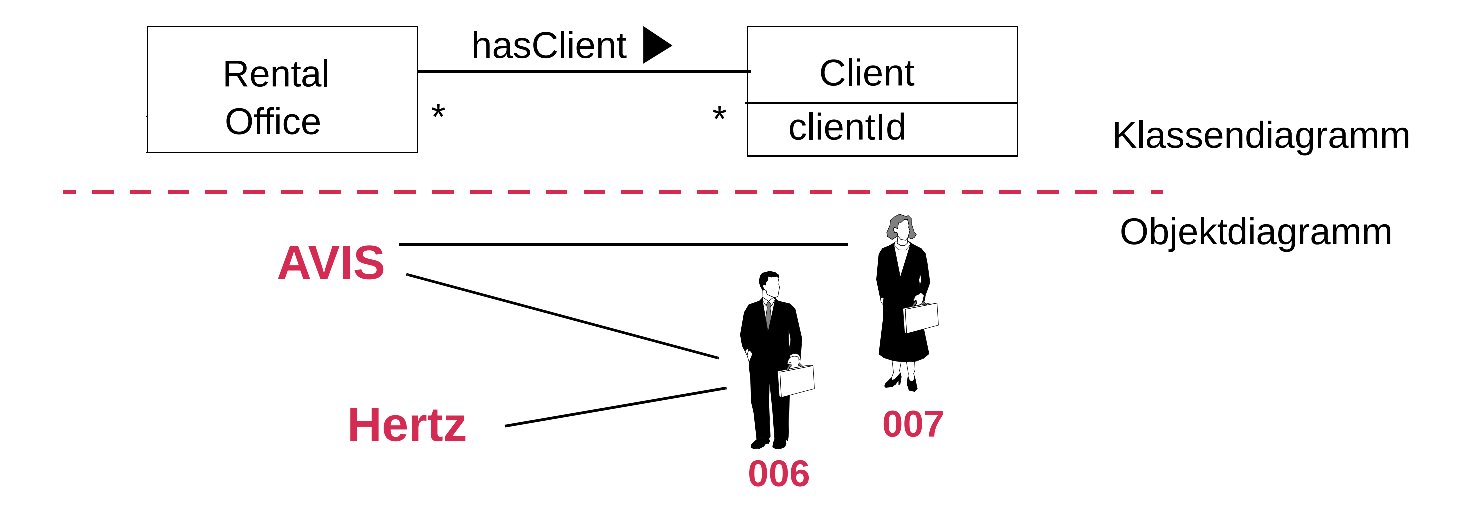
Verwendung eines Arrays nur bei feststehender maximaler Kardinalität geeignet

```
class Client {
...
private OrderedSet has_RentalContract;
...
}
```

besser Verwendung einer (geordneten) Menge - auch bei feststehender maximaler Kardinalität, da änderbarer



Probleme bei Assoziationen mit Eigenschaften:



- ⇨ da clientId ein Attribut der Klasse Client ist, hat jeder Kunde genau eine ihn identifiziernde Nummer bzw. Id
- ⇨ ist ein Client ein Kunde von mehreren RentalOffices, dann muss er trotz-dem bei allen die selbe Id besitzen
- ⇨ die clientId eines Kunden muss also Bestandteil bzw. Eigenschaft der Beziehung zu einem bestimmten RentalOffice werden



Einsatz von Assoziationsklassen:

Person

Company

Contract
Attributes
salary:Currency

super
sub
subcontract

super
sub
subcontract

Achtung:
Contract ist eine
binäre Assoziation
mit einem Attribut

{if Person has Contract C with Company A and
Contract C' with Company A' and
C is subcontract of C'
then
Company A has subcontract with Company A'}

- Assoziation mit Klasseneigenschaften (Link mit Objekteigenschaft)
- dürfen Attribute besitzen und Assoziationen eingehen
- zu jedem Objektpaar höchstens ein Linkobjekt (einer Assoziation), falls „unique“



Konsistenzbedingungen für Assoziationsklasse am Beispiel:

Contract

Dead Fish Inc.

Helga

Contract

Contract

super

subcontract

Arne

Contract

subcontract

Contract

sub

Contract

super

Hire&Fire

subcontract

sub

German Bank



Zugehörigkeit von Objekten zu Klassen:

- jedes Objekt ist **Instanz** genau einer (seiner) Klasse (normale UML-Interpretation)
- **abstrakte** Klassen (kursive Namen) besitzen keine Instanzen
- jedes Objekt ist **Mitglied**
  - ⇨ seiner eigenen Klasse
  - ⇨ und aller ihrer Oberklassen
- die **Extension** einer Klasse ist die Menge ihrer Mitglieder

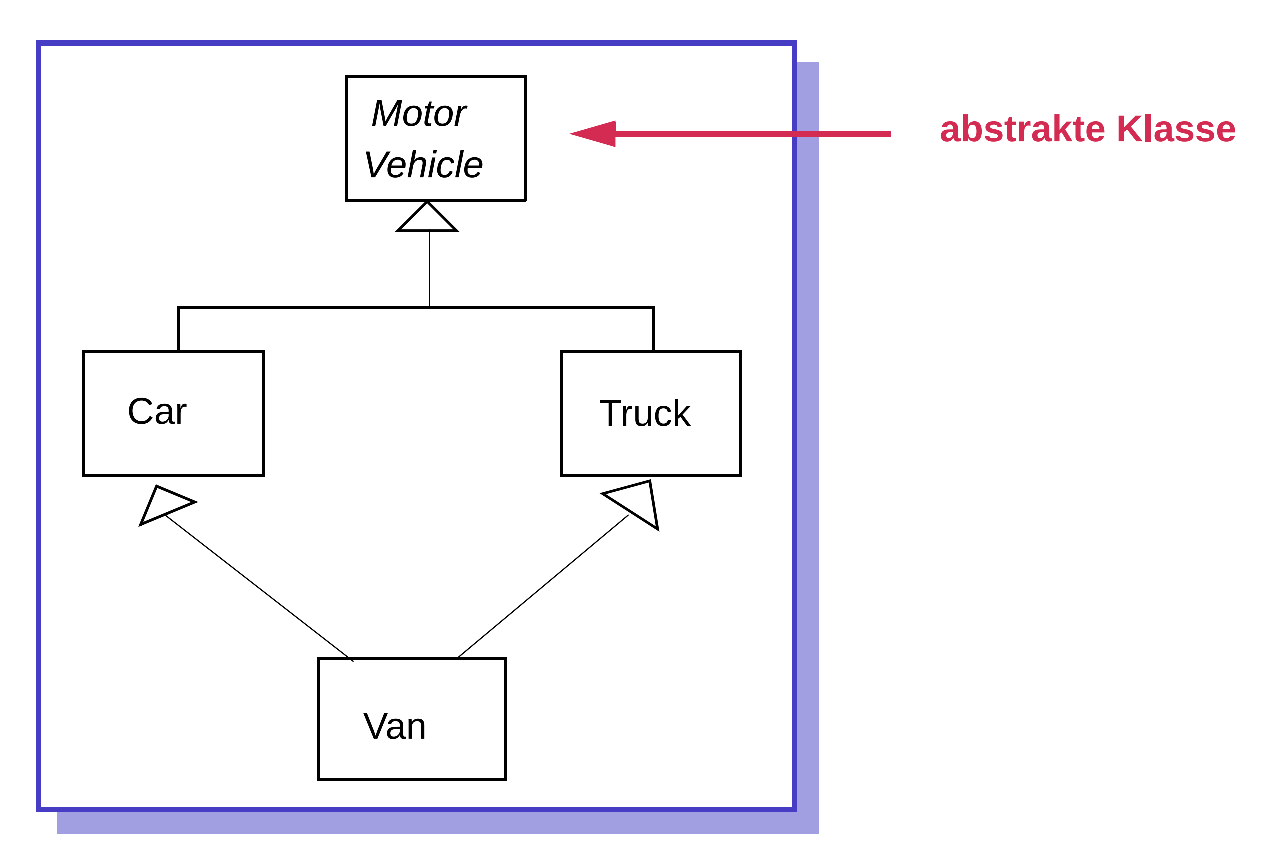
Grafik für Extensionen:

Extension der Menge der LegalEntity-Objekte

Extension der Menge der Company-Objekte	Extension der Menge der Person-Objekte



Mehrfachvererbung (multiple inheritance):



Achtung: Mit Mehrfachvererbung sehr vorsichtig umgehen. Oft handelt es sich dabei um einen Modellierungsfehler.



Einschub - Verschiedene Darstellungen von Interfaces und Nutzung:

klassenartige Notation

Visual Paradigm for UML Standard

Client

Import

<<use>>

<<Interface>>
SupplierInterface
+anOperation()

realize
(implements)

Supplier

alte „Lollipop"-Notation

Client

<<uses>>

SuppplierIn

Lollipop

<<realize>>

Supplier

neue UML 2.0 Notation

Visual Paradigm for UML Standard Edition(TU

Client

required
Interface

SupplierInterface

provided
Interface

Supplier



Erläuterungen zu Schnittstellen (Interfaces) von Klassen:

- Schnittstellen (**Interfaces**) spielen in UML die selbe Rolle wie in Java: man kann getrent von der Implementierung von Methoden in Klassen erst mal Operationen aufführen, die implementiert werden müssen
- eine Klasse kann mehrere Schnittstellen anbieten (implementieren), eine Schnittstelle kann durch mehrere Klassen implementiert werden; damit bietet die Klasse diese Schnittstellen an (**provided interfaces**)
- zusätzlich kann eine Klasse andere Schnittstellen benutzen (deren Existenz fordern), anstatt direkt die Methoden einer anderen Klasse zu benutzen (importieren); man spricht in diesem Fall von benötigten Schnittstellen (**required interface**)
- zueinander passende benötigte und angebotene Schnittstellen können verbunden werden; die benötigte und die angebotene Schnittstelle müssen entweder gleich sein oder die angebotene Schnittstelle eine Spezialisierung der benötigten.
- auf Schnittstellen gibt es Generalisierungshierarchien (Mehrfachvererbung); eine Schnittstelle erbt alle Operationen der spezialisierten Schnittstellen



Simulation der Mehrfachvererbung mit Interfaces:

Visual Paradigm for UML Standard Edition(TU Darmstadt)

<<Interface>>
MotorVehicle

Interface besitzt keine Implementierung, definiert nur Operationen für Klassen

CarImpl

<<Interface>>
Car

<<Interface>>
Truck

TruckImpl

Klasse implementiert Interface-Operationen

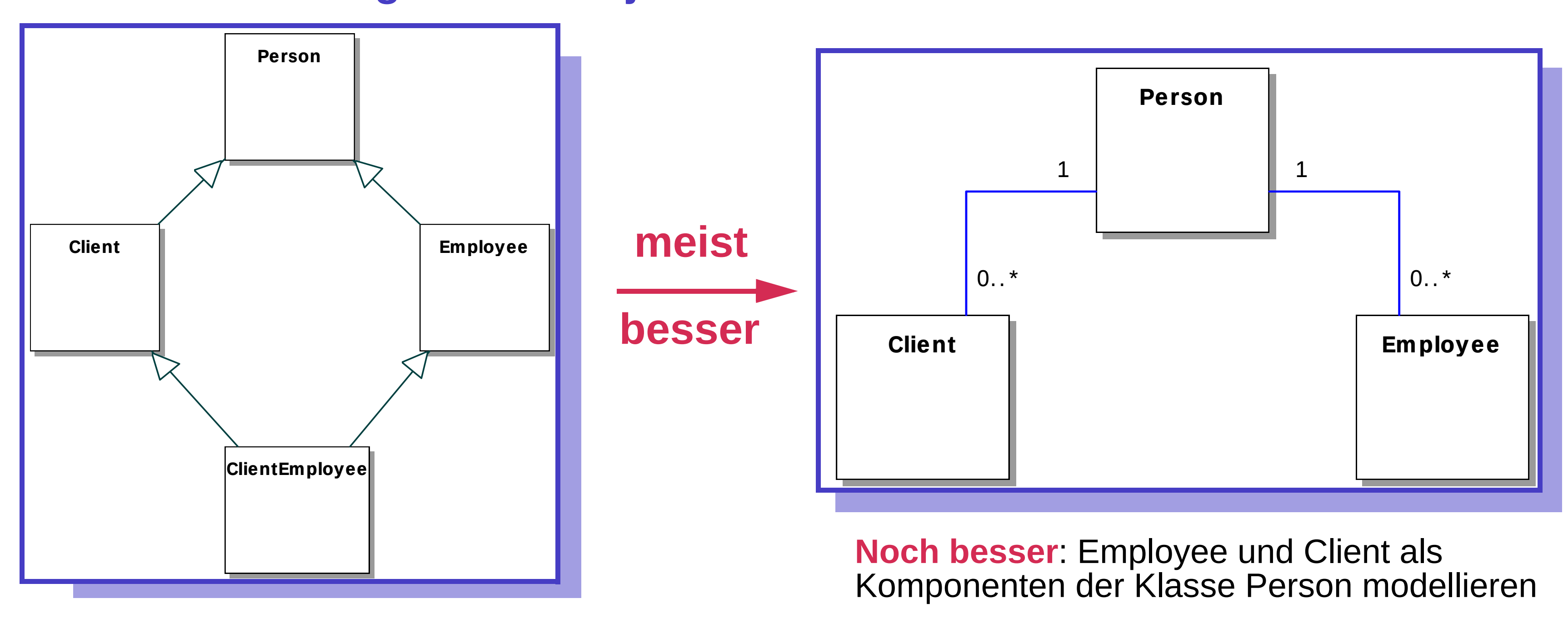
<<Interface>>
Van

VanImpl

- ☹ Simulation in Java notwendig (da keine Mehrfachvererbung für Klassen)
- ☹ Vererbung von Code (von MotorVehicle auf Car auf Van) nicht möglich
- ☹ mehr zum Einsatz von Interfaces (für Klassen) im folgenden Kapitel



Mehrfachvererbung versus Objektrollen:

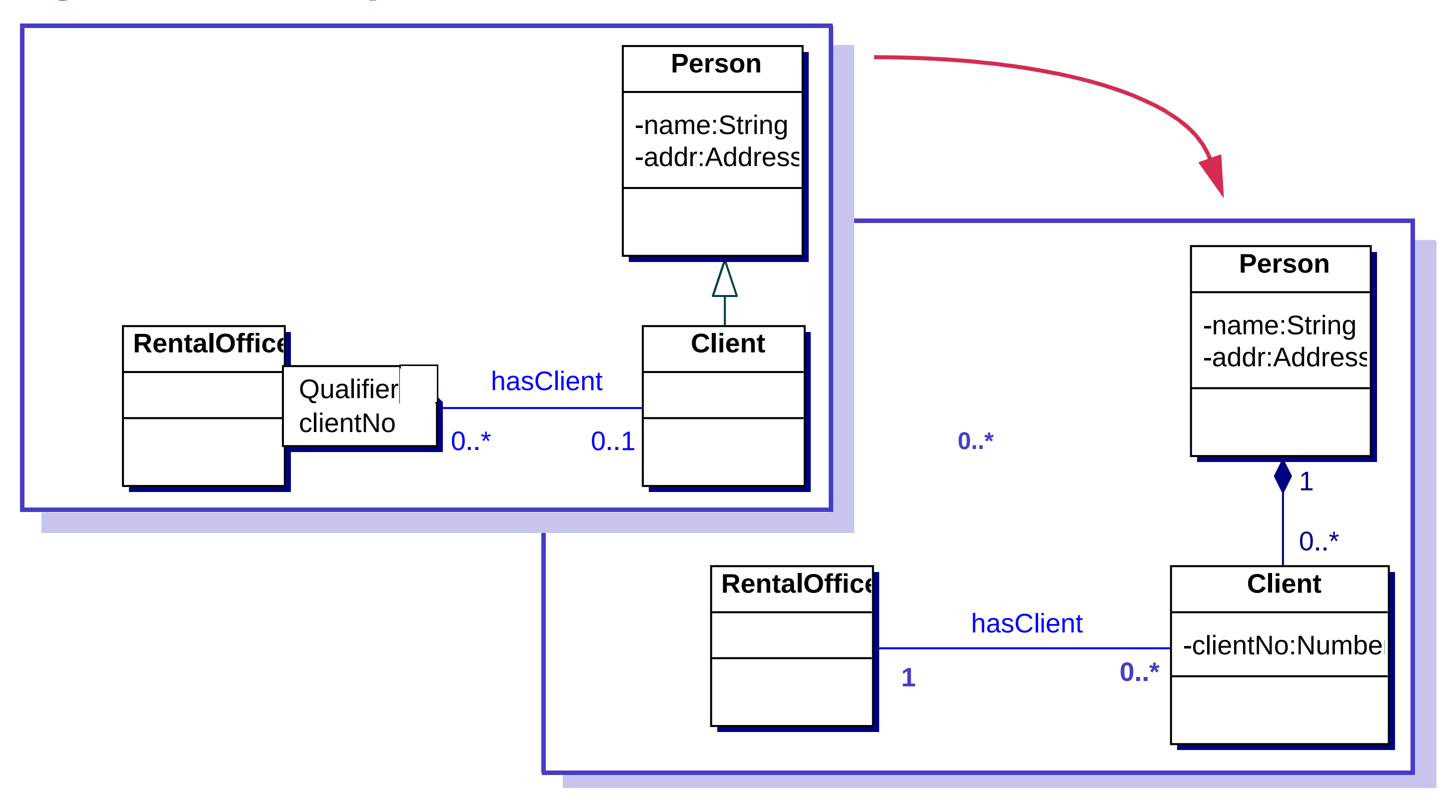


Noch besser: Employee und Client als Komponenten der Klasse Person modellieren

Probleme mit „linker“ Lösung:



Vererbung versus Komposition:

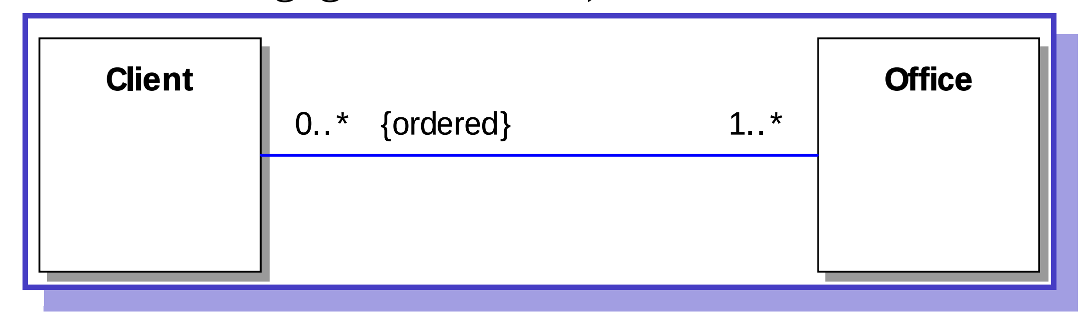


- ☺ bei Implementierung problematische Qualifizierung von hasClient eliminiert
- ☺ Client-Rolle „richtig“ als eigene Klasse implementiert
- ☹ Eigenschaften einer Person über mehrere Objekte „verstreut“



Weitere Eigenschaften von Assoziationsenden (wie bei Attributen):

- **readonly**: Links dürfen nur zusammen mit der Erzeugung des beteiligten Objekts angelegt werden; der Defaultwert ist false
- **unique**: es gibt keine zwei Links zwischen den selben Objekten; der Default ist true. Wird der Wert auf false gesetzt, so sind zwei verschiedene Links zwischen den selben beiden Objekten erlaubt (in Praxis wird immer mit Default gearbeitet)
- **ordered**: die Menge der Links, die von einem Objekt ausgehen (zu dem geordneten Assoziationsende hin) besitzen eine Ordnung (wie diese berechnet wird, kann nur als Kommentar angegeben werden); der Default ist false



- **derived**: die Assozation wird durch Formel aus anderen Assozationen berechnet; zur Kennzeichnung wird „/“ dem Namen vorangestellt; Default ist false



7.2 Spezifikation von Integritätsbedingungen mit OCL

Für die präzise Beschreibung von Konsistenzbedingungen (Integritätsbedingungen bzw. Constraints) auf Klassendiagrammen für

- ⇨ Attributwerte und Links von Objekten
- ⇨ Bedingungen für den Aufruf von Methoden
- ⇨ Auswirkungen des Aufrufs von Methoden

wurde die so genannte „**O**bject **C**onstraint **L**anguage“ **OCL** entwickelt, die eine (angeblich gut lesbare) Java-ähnliche Syntax für prädikatenlogische Ausdrücke anbietet. Es handelt sich dabei um eine **dreiwertige Logik** mit den Werten

true, false, undefined

(der Wert undefined steht für „unbekannter Wert“) und Rechenregeln der Form

true and undefined = undefined	false and undefined = false
true or undefined = true	false or undefined = undefined

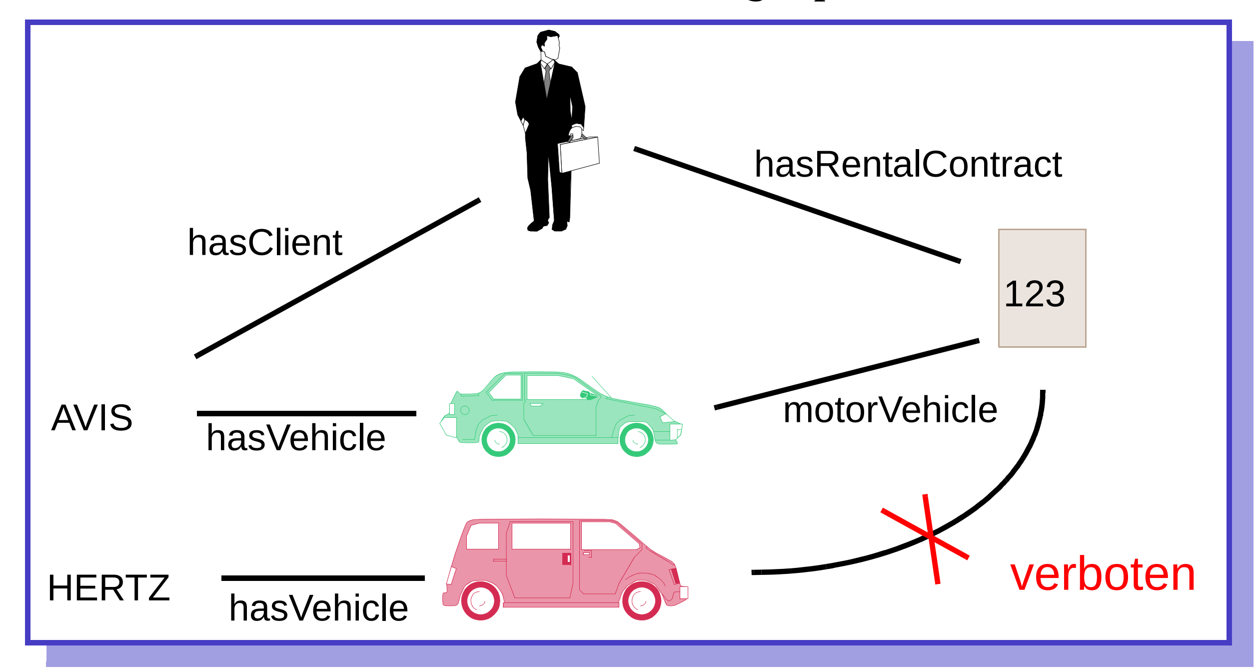


Verfeinerung von Klassendiagrammen um Konsistenzbedingungen:

Nur wenige Konsistenzbedingungen lassen sich durch die bisherigen Sprachmittel ausdrücken und wurden deshalb bislang umgangssprachlich formuliert (siehe Seite 341).

Beispiel:

"Kunde kann nur Autos mieten, die zum Wagenpark seiner Autovermietung gehören"





Die Object-Constraint-Language OCL:

Mit OCL lassen sich **Integritätsbedingungen (Invarianten)** formulieren, die

- ⇨ aus der Sicht einer Klasse (eines Objekts) beschrieben werden können (keine symmetrischen Bedingungen über mehreren Objekten)
- ⇨ nur auf andere Klasseninstanzen Bezug nehmen, die über Assoziationen mit dem "Hauptobjekt" verbunden sind
- ⇨ sich durch Prädikatenlogik ausdrücken lassen (keine Zeitbedingungen, …)

Beispiel von oben:

"Kunde kann nur Autos mieten, die zum Wagenpark seiner Autovermietung gehören"

Formulierung in OCL aus Sicht der Klasse Client:

```
inv:
  self.rentalOffice.motorVehicle->includesAll(self.rentalContract.motorVehicle)
```



Erläuterungen zum vorigen OCL-Ausdruck:

- **inv** = Invariant = steht für Invariante = immer geltende Boolesche Bedingung für alle Objekte der Klasse Client (in diesem konkreten Fall); die Bedingung wird für jedes Objekt der Klasse (alle Mitglieder der Klasse und ihrer Unterklassen) geprüft
- **self** entspricht in Java dem this und greift auf das gerade betrachtete Objekt der Klasse Client zu
- rentalOffice muss man sich als Methode der Klasse Client vorstellen, die alle über einen Link mit dem betrachteten Kunden verbundenen Objekte der Klasse RentalOffice zurückliefert
- entsprechend ist motorVehicle eine Methode der Klassen RentalOffice und RentalContract sowie rentalContract eine weitere Methode der Klasse Client
- Collection->... entspricht immer einem Methodenaufruf auf einer Kollektion (Set, Sequence, Bag) von Objekten
- includesAll wird auf einer Kollektion mit einer zweiten Kollektion als Parameter aufgerufen und liefert genau dann true zurück wenn die zweite Kollektion vollständig in der ersten Kollektion enthalten ist



Verschiedene Arten von OCL-Ausdrücken:

Jeder OCL-Ausdruck besitzt entweder eine Klasse oder eine Operation oder ein Attribut bzw. eine Assoziation als **Kontext**. Es gibt folgende Arten von OCL-Ausdrücken, die entweder direkt an eine Klasse, Methode oder Attribut/Assoziation als Kommentar in einem Diagramm angeheftet sind oder aber separat in einer Textdatei mit explizit angegebenem Kontext definiert werden:

1. **Invarianten zu Klassen**: Boole'sche Bedingungen, die auf allen Instanzen einer Klasse immer erfüllt sein müssen.

   context Client **inv**: age >= 18 -- oder: **self**.age >= 18

2. **Definition von Hilfsattributen und -operationen**: in mehreren OCL-Ausdrücken benötigte Teilausdrücke können auf diese Weise ausgelagert definiert werden ohne dass sie damit Bestandteil der Attribute und Operationen der Klasse (im Klassendiagramm) werden.

   context Client **def**: rentedVehicles = rentalContract.motorVehicle
   checkAgeLimit(limit:Integer) = (age >= limit)



Verschiedene Arten von OCL-Ausdrücken - Fortsetzung:

3. **Definition abgeleiteter und initialer Attributwerte**: so werden die initialen Werte von normalen Attributen und Berechnungsvorschriften für abgeleitete Attribute definiert, die in einem Klassendiagramm definiert sind (verwendet man statt Attributnamen die Namen der Rollen von Assoziationen, so können auch Initialwerte für Assoziationen und abgeleitete Assoziationen definiert werden):

```
context Client::age : Integer    init:  0
    -- das Alter einer neu erzeugten Person wird auf 0 gesetzt

context Client::noVehicles : Integer  derive:
    rentalContract->size()
    -- die Anzahl der (ausgeliehenen) Fahrzeuge einer Person entspricht
    -- der Anzahl der Ausleihverträge dieser Person

context RentalOffice::company : Company   derive:
    client.company
    -- definiert eine abgeleitete Assoziation mit Rollenname „company“
    -- als Abkürzung für den Pfad „client.company“
```



Verschiedene Arten von OCL-Ausdrücken - Fortsetzung:

4. **Vor- und Nachbedingungen für Operationen**: für eine Operation einer Klasse (oder Schnittstelle) kann man angeben unter welchen Bedingungen sie nur aufgerufen werden darf (Vorbedingungen) und welche Bedingungen nach ihrer erfolgreichen Ausführung immer gelten:

```
context Client::incrementFrequentRenterBonus(bonus:Integer):Integer
    pre:   bonus > 0 and currentBonus+bonus <= 50
           -- es ist nur eine Bonuserhöhung auf maximal auf 50
    post:  currentBonus = bonus + currentBonus@pre and
           result = currentBonus
           -- mit @pre wird auf den Wert des Attributs currentBonus vor
           -- der Ausführung der Operation zugegriffen
```

Achtung: ist beim Aufruf einer Operation die Vorbedingung verletzt oder nach Ausführung einer Operation ihre Nachbedingung, so ist das Ausführungsverhalten undefiniert. Eine Implementierung kann dann beispielsweise eine Ausnahme „werfen“ oder den Aufruf der Methode ignorieren oder ... .



Interaktion von Vererbung/Generalisierung und OCL-Ausdrücken:

- Invarianten, Vor- und Nachbedingungen gelten für **alle Mitglieder** einer Klasse (also auch für die Instanzen ihrer Unterklassen)
- Instanzen einer Unterklasse müssen **alle Invarianten** ihrer Oberklassen erfüllen plus ggf. zusätzliche eigene „schärfere" Invarianten
- eine Unterklasse darf die **Vorbedingungen** für geerbte Methoden „**aufweichen**" aber nicht einschränken (Methoden müssen unter den geerbten Vorbedingungen weiterhin aufrufbar sein)
- eine Unterklasse darf die **Nachbedingungen** für geerbte Methoden „**verschärfen**" aber nicht aufweichen (Methoden müssen die Nachbedingungen geerbter Methoden weiterhin erfüllen); es werden dabei zusätzliche Eigenschaften der Ausgabewerte oder Attributwerte nach Ausführung der Methode festgelegt
- **Initialisierungen** von Eigenschaften eines Objekts sind wie Nachbedingungen von Konstruktoren zu behandeln
- Ausdrücke für **abgeleitete Eigenschaften** (Attribute und Assoziationen) sind wie Invarianten von Klassen zu behandeln



Zulässige Verfeinerung von OCL-Bedingungen bei Vererbung:

context Client1::incrementFrequentRenterBonus(bonus:Integer):Integer
-- Client1 sei eine direkte Unterklasse von Client

pre: bonus >= 0 **and** currentBonus+bonus <= 100
-- mit bonus = 0 soll nun Bonus auf 0 zurückgesetzt werden
-- zusätzlich dürfen Client1-Objekte Bonus bis 100 besitzen

post: if bonus = 0 then
current Bonus = 0
-- für neu erlaubten Parameter-Wert neue Nachbedingung
else
currentBonus = bonus + currentBonus@pre

post: result = currentBonus

context Client1
inv: age >= 21
-- weitergehende Einschränkung des Alters (statt >= 18)



Unzulässige Verfeinerung von OCL-Bedingungen bei Vererbung:

context Client2::incrementFrequentRenterBonus(bonus:Integer):Integer
-- Client2 sei eine direkte Unterklasse von Client

pre: bonus > 0 **and** currentBonus+bonus <= 30
-- unzulässige Einschränkung der Vorbedingung

post: currentBonus <= bonus + currentBonus@pre
-- unzulässige Aufweichung der Nachbedingung

post: result = currentBonus

context Client2
inv: age >= 17
-- unzulässige Lockerung der Altersvorschrift (statt >= 18)



OCL-Navigationsausdrücke:

context Client

inv: self.age >= 18 -- alle Kunden sind mindestens 18 (**Attributzugriff**)

---

context RentalOffice

... self.myClients ... -- Kollektion aller Kunden (**Linktraversierung**)
(Angabe des Rollennamens)

---

context Client

... self.rentalOffice ... -- liefert ein Objekt zu einer Person (**Linktraversierung**)
(Angabe des Klassennamens
mit kleinem Anfangsbuchstaben)

---

context RentalOffice

... self.myClients.age ... -- Kollektion aller Altersangaben aller Kunden



Weitere OCL-Features - prädikatenlogische Boole'sche Ausdrücke:

- die aussagenlogischen Operatoren not, or, and, implies für logische Negation, Disjunktion, Konjunktion und Implikation (Schlussfolgerung) sowie if-then-else mit Boole'scher Bedingung
- Allquantifizierung der Prädikatenlogik mit forAll:

```
context RentalOffice                    -- Kundennamen sind eindeutig
  inv: self.myClients->forAll(c1, c2 |
                         (c1 <> c2) implies (c1.name <> c2.name) )
```

- Existenzquantifizierung der Prädikatenlogik mit exists:

```
context RentalOffice                    -- Kundennamen sind eindeutig
  inv: not self.myClients->exists(c1, c2 |
                         (c1 <> c2) and (c1.name = c2.name) )
```



Erläuterungen zu Ausdrücken mit Existenz- und Allquantoren:

- mit „exp.operation"-Notation wird in der Regel eine Operation auf das element-wertige Ergebnis eines Ausdrucks angewendet (z.B. alle Kunden genau einer Firma bestimmen)
- mit „exp->operation"-Notation wird immer eine Operation auf ein mengen-wertiges Ergebnis eines Ausdrucks angewendet (z.B. auf der Menge aller Kunden eine bestimmte Eigenschaft überprüfen)
- mit den Variablen c1 und c2 werden jeweils alle Elemente der betrachteten Menge in allen Kombinationen durchlaufen (c1 und c2 können also auch auf das selbe Objekt verweisen); für alle Kombinationen wird der Ausdruck hinter „|" überprüft
- der Ausdruck forAll(c1, c2 | (c1 <> c2) implies (c1.name <> c2.name)) überprüft, ob für alle Paare verschiedener Kunden (c1 ungleich c2) deren Namen ungleich sind
- der Ausdruck exists(c1, c2 | (c1 <> c2) and (c1.name = c2.name)) überprüft, ob es ein Paar verschiedener Kunden gibt, bei denen die Namen gleich sind



Weitere OCL-Features - Standarddatentypen Integer, Float und String:

- **Standarddatentyp Integer**: die Konstanten 0, 1, -1, ... sind mit den üblichen Operatoren darauf definiert (+ , - , * , / , abs() , < , ...)
- **Standarddatentyp Float**: die Konstanten 0.0, 1.1, -1.1, ... sind mit den üblichen Operatoren darauf definiert (+ , - , * , / , abs() , < , ...)
- **Standarddatentyp String**: Konstanten der Art 'ich bin ein String' sowie die folgenden Operatoren sind definiert:
  - s1.concat(s2): konkateniert zwei Strings
  - s1.size(): gibt die Länge eines Strings zurück
  - s1.substring(i1, i2) gibt den entsprechenden Teilstring zurück; dabei muss 1 <= i1 <= i2 <= s1.size() gelten, sonst wird Ergebnis undefiniert
- selbst definierte **Aufzählungstypen** wie Colour mit den Literalen white, ... können in OCL-Ausdrücken wie folgt verwendet werden:

```
Colour::white     -- so wird das Literal notiert
```



Typ-Überprüfungen, - Einschränkungen und undefinierte Ausdrücke:

- Mit obj.oclIsTypeOf(Class) kann man überprüfen, ob ein Objekt direkte Instanz einer bestimmten Klasse ist (liefert true oer false als Ergebnis).
- Mit obj.oclIsKindOf(Class) kann man überprüfen, ob ein Objekt direkte Instanz einer bestimmten Klasse oder einer ihrer Unterklassen (oder Interface) ist.
- Mit obj.oclAsType(Class) kann man einen Ausdruck, der die Klasse CA als Typ hat, auf die Klasse Class „einschränken", falls Class eine direkte oder indirekte Unterklasse von CA ist. Wird der Operator zur Laufzeit auf eine Instanz einer Klasse CI angewendet, die keine direkte oder indirekte Unterklasse von Class ist, dann ist das Ergebnis undefiniert.
- im Prinzip kann jeder Ausdruck den Wert „**undefiniert**" annehmen; das lässt sich mit dem Prädikat oclIsUndefined überprüfen; so gilt etwa:
  not obj.oclAsType(Class).oclIsUndefined() = obj.oclIsKindOf(Class)
- der Wert „undefiniert" propagiert (fast immer) durch Ausdrücke hindurch;
  Ausnahmen: true = undefined or true sowie false = undefined and false sowie e = if true then e else undefined.



Verschiedene Arten von Kollektionen:

- **Set** (unordered, unique elements): die „normale" Menge ohne Ordnung auf den Elementen und ohne Duplikate
- **OrderedSet** (ordered, unique elements): eine Liste von Elementen, die gemäß Index der Elemente sortiert ist; es gibt keine Duplikate
- **Bag** (unordered, not unique): eine ungeordnete „Ansammlung" von Elementen, die unsortiert sind, oft Multimenge genannt
- **Sequence** (ordered, not unique): eine Liste von Elementen, die gemäß Index der Elemente sortiert ist und Duplikate enthalten darf

Operationen auf Kollektionen:

- die üblichen Mengenoperationen (Vereinigung, ...)
- Operationen zur Iterationen über Mengen, Selektion von Teilmengen, ...
- Operationen für Zugriff auf einzelne Listenelemente über Index



Beispiele für Standard-Operationen auf Kollektionen:

- weitere Operationen auf Kollektionen (Mengen, Multimengen, Listen)
  - ⇨ z.B. s->includes(e) für Element e ist in einer Kollektion s enthalten

```
context Client
    inv: self.rentalContract.motorVehicle->forAll( v |
           self.rentalOffice.motorVehicle->includes(v))
```

  - ⇨ z.B. s1->includesAll(s2) für alle Elemente von s2 sind enthalten in s1

```
context Client
    inv: self.rentalOffice.motorVehicle->includesAll(
              self.rentalContract.motorVehicle)
```

  - ⇨ mit union, intersection Operationen für Vereinigung und Durchschnitt



Operationen zur Iteration auf Mengen und Selektion von Elementen:

- Die **Iteration** über einer Menge setzt zunächst eine „Akkumulator“-Variable auf einen Initialwert und berechnet dann für jedes Element einer gegebenen Kollektion den Wert auf Basis des bisherigen Wertes neu:

```
context RentalOffice def:
    valueOfAllCars = motorVehicle->iterate(
                        vehicle: Vehicle; totalValue: Integer = 0 |
                        vehicle.value + totalValue )
    -- der Preis aller Fahrzeuge der Firma wird berechnet
```

- Die **Auswahl** von Elementen aus einer Kollektion, die eine Bedingung erfüllen:

```
context RentalOffice def:
    oldClients = client->select(age >= 60)
```

- Die **Elimination** von Elementen aus einer Kollektion, die eine Bedingung erfüllen:

```
context RentalOffice def:
    oldClients = client->reject(age < 60)
```



Behandlung von Kollektionen bei Navigation:

- **self**.rentalOffice liefert aufgerufen auf einem Objekt der Klasse Client eine Kollektion (Sequenz) von Rental-Office-Objekten zurück
- rentalOffice.motorVehicle ist eine abkürzende Schreibweise für **self**.rentalOffice->collect(motorVehicle) bzw. für **self**.rentalOffice->collect(anOffice | anOffice.motorVehicle)
- collect wird auf einer Kollektion von Elementen aufgerufen mit einem Ausdruck als Argument: cs->collect(e) ist die Vereinigung der Auswertung des Ausdrucks e für jedes einzelne Objekt in der Kollektion cs
- ein OCL-Ausdruck der Art obj.role1.role2. ... bildet also automatisch die Vereinigung der Ergebnisse der Anwendung von role2 auf alle Ergebnisse des Aufrufs von role1 auf obj
- Ergebnisse sind immer Listen (Sequenzen), die mit dem Aufruf list->asSet() in eine Menge ohne Duplikate umgewandelt werden können



OCL-Constraints im Klassendiagramm:

```
{inv: self.client->forAll(c1, c2 |
        c1 <> c2 implies c1.name <> c2.name)}
```

```
{inv:
self.rentalOffice.motorVehicle->includesAll(self.rentalContract.motorVehicle)}
```

RentalOffice
Attributes
name:Integer
/noOfCars:Integer

```
{derive: motorVehicle->size()}
```

1..* rentalOffice — client 0..*

Client
Attributes
name:String
age:Integer

```
{inv:
self.age >= 18}
```

normale Invariante; muss immer gelten

rentalOffice 1 — motorVehicle 1..*

client 1 — rentalContract 0..*

MotorVehicle
Attributes
age:Integer
price:Integer

```
{init: 0}
```

Initialisierung eines Attributs (age)

1 motorVehicle — rentalContract 0..*

RentalContract
Attributes
/deposit:Integer

```
{derive:
if motorVehicle.age < 10 then motorVehicle.price * (10 - motorVehicle.age) div 100 else 0 endif
-- für ein neues Auto wird ein Zehntel des Neupreises als Kaution verlangt, für ein 10 Jahre altes Auto nichts}
```

Definition eines abgeleiteten Attributs (deposit); dieses besitzt immer diesen berechneten Wert



Zusammenfassung der Verwendung von OCL:

- OCL ist eine umfangreiche Sprache basierend auf der Prädikatenlogik, die nur in Auszügen hier vorgestellt wurde
- **Invarianten** beziehen sich immer auf eine Klasse und müssen für alle Instanzen dieser Klasse zu allen Zeiten gelten
- mit **derive** kann man den Wert eines abgeleiteten Attributes durch eine Formel festlegen; bei jedem lesenden Zugriff auf das Attribut wird die Formel berechnet
- mit **init** kann man einen initialen Wert für ein Attribut festlegen, der später durch Zuweisungen überschrieben werden kann
- des weiteren kann OCL dazu verwendet werden, **Vor- und Nachbedingungen** an Methoden zu formulieren, die vor dem Aufruf und nach der Durchführung einer gerufenen Methode gelten müssen
- OCL wird manchmal auch in anderen Diagrammarten für die **Berechnung von Werten** und die Definition von **Bedingungen** eingesetzt



7.3 Verfeinerte Verhaltensbeschreibung (UML Interaction Diagrams)

Aufgabe:

Mit Interaktionsdiagrammen werden die Anwendungsfälle aus Abschnitt 5.2 genauer beschrieben, sobald die dabei benötigten Klassen und ihre Operationen klar sind. Die mit Abstand populärste Form von Interaktionsdiagrammen (Interaction Diagrams) sind die **Sequenzdiagramme** (andere Formen werden hier nicht vorgestellt).

Ergebnisvariable

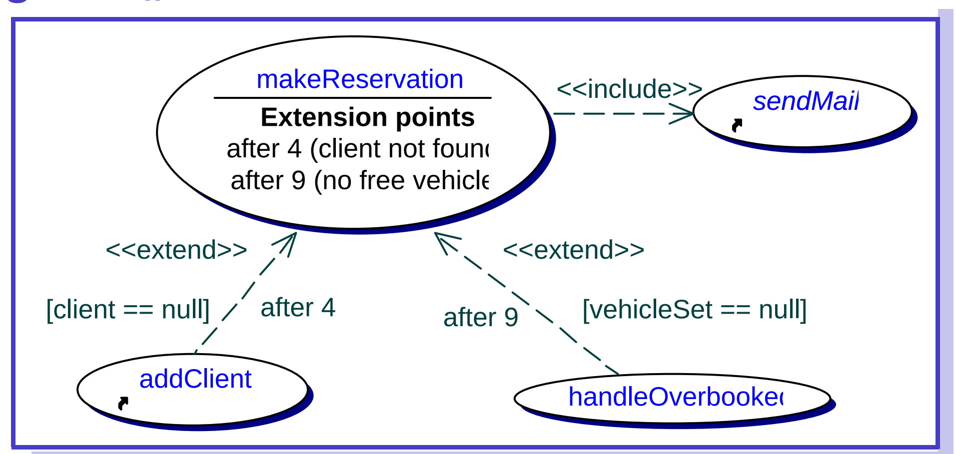
Laufvariable

Vorgehensweise:

- ⇨ zuerst Anwendungsfälle mit textueller Beschreibung (oder ggf. Aktivitäts-diagramme) in einfache Sequenzdiagramme übersetzen, die Aussenverhal-ten eines Systems beschreiben
- ⇨ dann Systemoperationen in Klassenoperationen zerlegen (bei der Fallbeispieldefinition mit Aktivitätsdiagrammen keine Voraussetzung)
- ⇨ nun mit Sequenzdiagrammen Nachrichtenaustausch innerhalb des Systems mit Betonung des zeitlichen Ablaufs modellieren



Unser Anwendungsfall „makeReservation“:



Clerk	MVRS
1. Aufruf von makeReservation	
	2. erfragt Kundenname n
3. gibt Name eines Kunden ein	
	4. sucht Client n in Datenbank
	5. erfragt Reservierungszeitraum p
6. gibt Zeitraum p ein	
	7. erfragt Fahrzeugkategorie c
8. gibt Fahrzeugkategorie c ein	
	9. bestimmt freie Fahrzeuge zu p und c
	10. erfragt gewünschtes Fahrzeug
10. wählt gewünschtes Fahrzeug aus	
	11. trägt Reservierung in Datenbank ein
	12. ruft Anwendungsfall sendMail



Präzisierung von Anwendungsfall mit einfachem Sequenzdiagramm:



Erläuterungen zu Sequenzdiagrammen:

- Sie beschreiben den zeitlichen Ablauf der Kommunikation von mehreren Objekten; die Lebenszeit der Objekte wird durch vertikale **Lebenslinien** definiert.
- **Aktive Objekte** (alle im vorigen Beispiel) haben einen durchgehenden dicken Balken als Lebenslinie (und in als Kasten doppelte vertikale Linien); externe Akteure eines Systems sind immer aktive Objekte.
- Die Reihenfolge der **Nachrichten** (horizontale Pfeile zwischen zwei Lebenslinien) in vertikaler Richtung bestimmt die Reihenfolge ihrer Abarbeitung.
- Pfeile mit offener Pfeilspitze stellen Nachrichten dar, die als **asynchrone Signale** von einem Sender zu einem Empfänger geschickt werden (Sender ist nicht blokkiert bis ein Ergebnis beim Empfänger berechnet und zurückgeschickt worden ist).
- Pfeile mit geschlossenen Pfeilspitzen stellen Nachrichten dar, die als **synchrone Aufrufe** des Senders von Operationen des Empfängers realisiert sind. In diesem Fall ist der Sender blockiert, bis die Operation ausgeführt wurde.
- Achtung: die Nachrichtenübertragung ist hier in der Vorlesung immer **zeitlos**; schräg verlaufende Nachrichtenpfeile defnieren zeitbehaftete Kommunikation.



Sequenzdiagramm mit passivem (internen) Objekt:

Visual Paradigm for UML Standard Edition(TU Darmstadt)

aClerk:Clerk

:MVRS

:DB

theClient:Client

makeReservation

getClient

returnClient(cn)

findClient(cn)

findClient(): theClient

getPeriod

returnPeriod(p)

getCategory

returnCategory(c)

findVehicles(p,c)

findVehicles(): vs

selectVehicle(vs)

returnSelectedVehicle(v)

addReservation(cn,p,c,v)

addReservation(): r

sendMail(r)

passives Objekt

aktive Zeit-spanne

Signal mit Parameter (Rückgabe-wert für voriges Signal)

Operations-aufruf mit Parametern

return von Operations-aufruf



Erläuterungen zu Sequenzdiagramm - Fortsetzung:

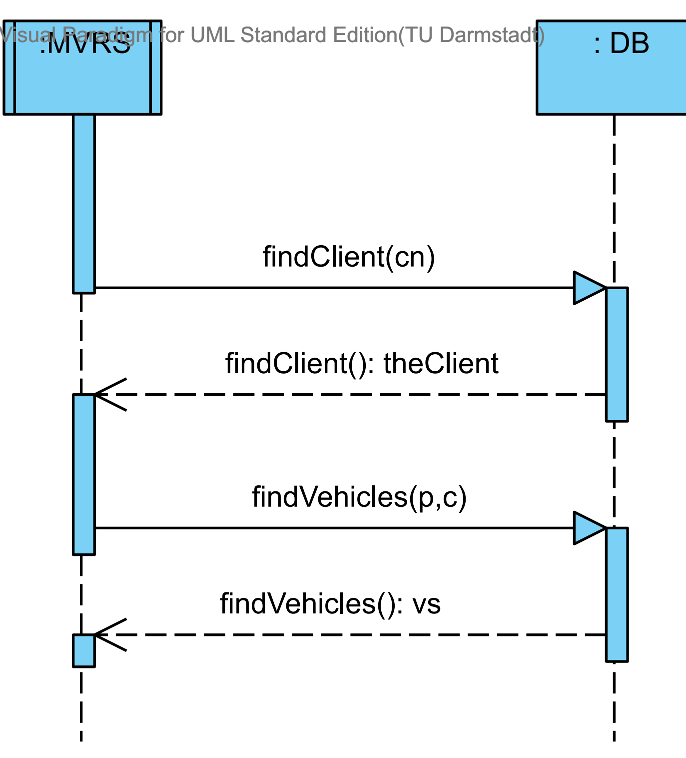
- Passive Objekte besitzen eine gestrichelte Lebenslinie, solange sie inaktiv sind; sie werden nur durch Empfang einer Nachricht (meist ein synchroner Operations-aufruf) aktiv (bis zum Ende der Ausführung der gerufenen Operation).
- Ein asynchrones Signal oder ein synchroner Operationsaufruf kann beliebig viele Parameter besitzen. Diese Pfeile sind wie folgt beschriftet:
  [<Var> =] <Operationsname> [(<Parameterliste>)]
- Ein synchroner Operationsaufruf wird mit einem gestrichelten „return"-Pfeil mit offener Spitze beendet; dieser Pfeil ist wie folgt beschriftet:
  <Operationsname> [(<Parameterliste>)] [: <Wert>]

Achtung:

Theoretisch könnten Signale (als Daten) auch synchron verschickt und Operationen auch asynchron aufgerufen werden, aber meist verwendet man nur asynchrone Signale und synchrone Operationsaufrufe.



Genauere (aber eher unübliche) Sequenzdiagrammdarstellung:



Diese Darstellung werden wir im Folgenden nicht weiter verwenden, sondern die Lebenslinien aktiver Objekte immer durchgehend „dick“ darstellen.

Aktive Objekte leihen ihre „dicke“ Lebenslinie beim Aufruf von Operationen an passive Objekte aus. Während der DB-Operationsaufrufe oben kann ja das MVRS-Objekt nichts anderes machen und ist deshalb zeitweise nicht aktiv. Das wird hier korrekt durch gestrichelte Lebenslinie dargestellt.



Grafisch einfachere Darstellung von Operationsaufrufen:

Visual Paradigm for UML Standard Edition(TU Darmstadt)

aClerk:Clerk

:MVRS

:DB

theClient:Client

makeReservation

getClient

returnClient(cn)

theClient = findClient(cn)

return-Pfeile von Operations-aufrufen werden nicht dargestellt

getPeriod

returnPeriod(p)

getCategory

returnCategory(c)

vs = findVehicles(p, v)

selectVehicle(vs)

returnSelectedVehicle(v)

r = addReservation(cn,p,c,v)

sendMail(r)



Akurate Darstellung rekursiver Operationsaufrufe:

der rekursive Aufruf wird aufwändiger grafisch dargestellt.



Behandlung von Sonderfällen in Sequenzdiagrammen:

Bedingter/optionaler Teil des Ablaufs mit Bedingung in [...]



Weitere Interaktionsoperatoren

Mit opt haben wir einen ersten Interaktionsoperator für Sequenzdiagramme kennengelernt, der sich über mehrere Lebenslinien erstreckt. Es gibt des weiteren:

- **loop(m,n)**: die umschlossenen Interaktionen werden mindestens m mal, höchstens n-mal ausgeführt (zusätzlich kann es Iterationsbedingung geben)
- **alt**: eine Reihe von Interaktionsbereichen (Alternativen) besitzen jeweils eine Bedingung und werden ausgeführt, falls diese Bedingung erfüllt ist (die Bedingungen sollten sich gegenseitig ausschließen)
- **par**: eine Reihe von Interaktionssequenzen können nebenläufig (entweder sequentiell verschränkt oder echt parallel) in beliebiger Reihenfolge ausgeführt werden
- **ref**: es wird eine separat definierte Interaktionssequenz aufgerufen (referenziert), die über mehrere Lebenslinien hinweg definiert sein kann
- ...



Beispiel für Schleife und Aufruf eines anderen Interaktionsdiagramms:

Visual Paradigm for UML Standard Edition(TU Darmstadt)

aClerk:Clerk

:MVRS

vs:Vehicle

findVehicles(p,c)

ok = false

loop (0, vs.size())

[not ok]

v = removeElement()

selectVehicle(v)

returnSelectVehicle(ok)

ref

CheckAndHandleOverbooked(v,ok)

Möglichkeit zur Definition einer einfachen Anweisung

Erzeugung einer Menge von Objekten vs

Schleife, die maximal so oft ausgeführt wird, wie vs Elemente enthält

Bedingung für Fortführung der Schleife

Aufruf eines anderen Interaktionsdiagramms



Zusammenfassung wichtiger Elemente von Sequenzdiagrammen:

Visual Paradigm for UML Standard Edition(TU Darmstadt)

Aktives Objekt: Class A

passives Objekt: Class B

Aktive Menge: Class C

Objekterzeugung

synchr. Operationsaufruf

Erzeugungsnachricht

Neues Objekt: Class D

rekursiver Aufruf

Return

loop

[Bedingung]

asynchrones Signal

Zerstörungsnachricht

Objektzerstörung

ref

Sequence Diagram2

Aufruf eines Sequenzdiagramms



Anmerkungen zu Sequenzdiagramme:

- Sequenzdiagramme werden **manuell** erstellt oder **generiert**
  - ⇨ vom Modellierer (manuell) oft in frühen Phasen der Softwareentwicklung
  - ⇨ von Werkzeugen (generiert) als Visualisierungen von Testläufen
- Hauptanwendungsgebiete von Sequenzdiagrammen sind:
  - ⇨ detaillierte Beschreibung von **Anwendungsfällen**
  - ⇨ Festlegung von **Testfällen** (können mit generierten Diagrammen bei Testläufen verglichen werden)
- **MSCs** = Message Sequence Charts (Diagramme) sind verwandter ITU-T Standard der Telekommunikationsindustrie mit etwa den selben Konstrukten
- Werkzeuge unterstützen Sequenzdiagramme sehr unterschiedlich umfangreich (ich kenne kein Werkzeug, das UML-Sequenzdiagramme vollständig realisiert)



7.4 Zustandsautomaten (UML Statecharts)

Aufgabe:

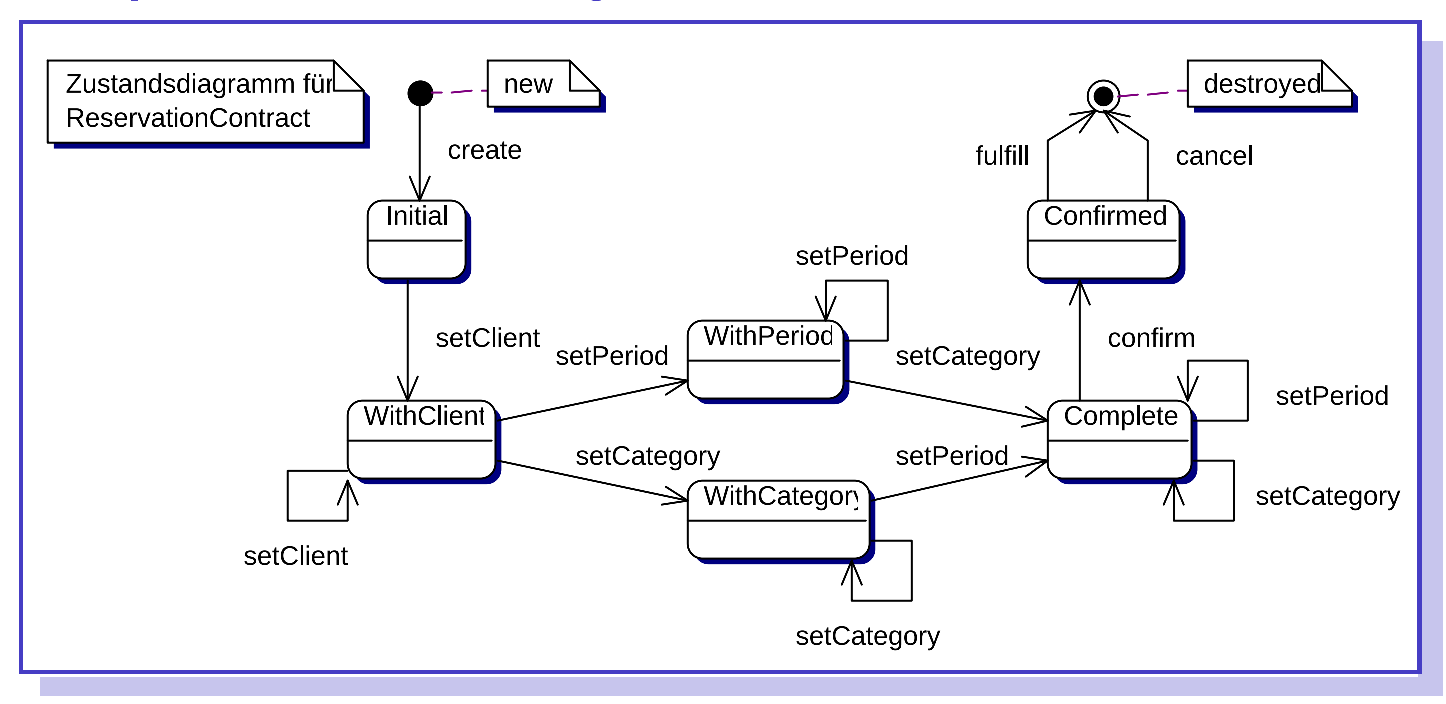
Beschreibung des **Verhaltens der Objekte einer Klasse**
(und nicht Zusammenspiel der Objekte verschiedener Klassen).
Es werden hierarchische Zustandsdiagramme benutzt, die eine
Variante von Harel's Statecharts sind [Ha87].

Vorgehensweise:

- ⇨ Identifikation semantisch sinnvoller Objektzustände
- ⇨ Zuordnung von Operationen zu Zuständen (welche Operationen können in welchem Zustand bearbeitet werden)
- ⇨ Bestimmung des „Objektlebenszyklus" als zulässige Zustandsfolgen (Operationsaufrufe/Signale als Ereignisse lösen Zustandsübergänge aus)
- ⇨ Zuordnung von Aktionen zu Zustandsübergängen
- ⇨ vollständige „Ausprogrammierung" der Objektimplementierung



Erstes Beispiel für Zustandsdiagramm:



- teilweise werden ähnliche Notationselemente wie bei Aktivitätsdiagrammen verwendet (für Markierung Anfangszustand, Endzustand, ...)
- UML-Zustandsdiagramme sind deterministische endliche Automaten mit einer Reihe zusätzlicher Schreibabkürzungen (übliche Bezeichnung: **FSM** = **F**inite **S**tate **M**achine)

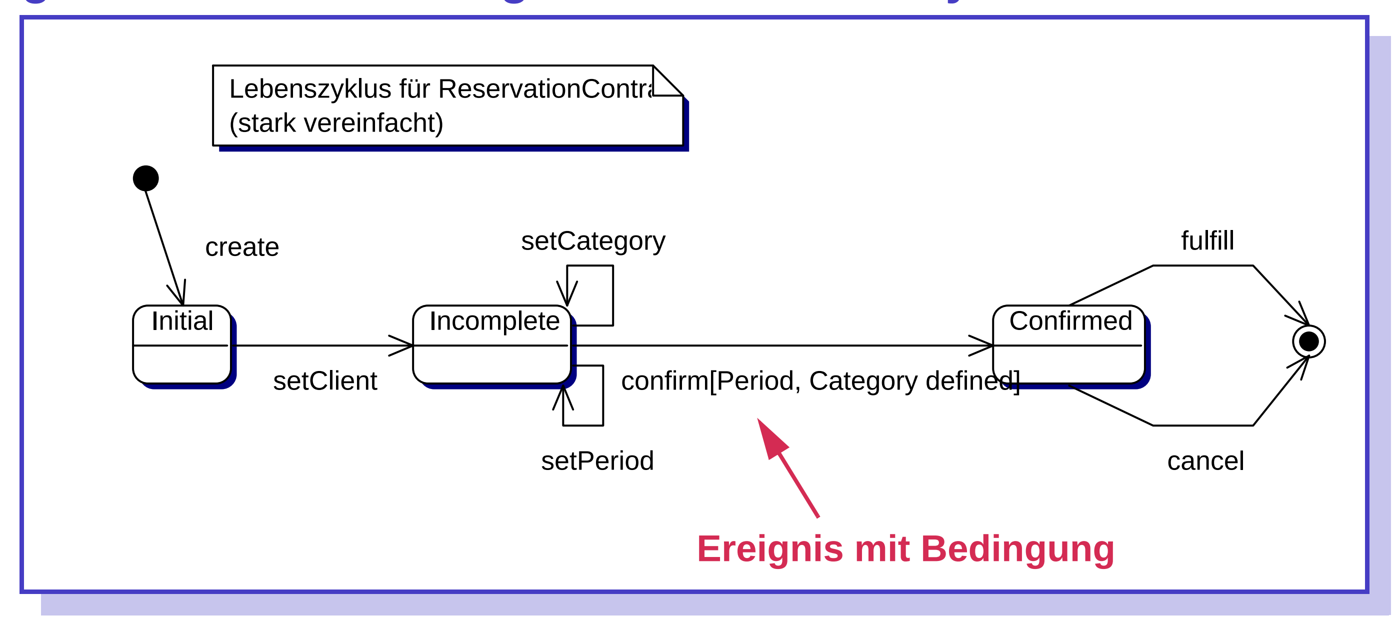


Anmerkungen zur FSM:

- die Operationen setVehicle und setDeposit wurden weggelassen, um das Diagramm überschaubar zu halten
- die Rechtecke sind die **Zustände** der FSM
- es gibt immer einen **Anfangszustand** und meist genau einen **Endzustand**
- die Pfeile definieren die Übergangsfunktion von einem Zustand zum nächsten, sie werden **Transitionen** genannt und sind mit Signalen beschriftet
- die Übergangsfunktion ist **partiell**, da man nicht mit jedem Signal (Ereignis) aus einem Zustand in einen anderen kommt
- die FSM ist **deterministisch**, da für jedes Paar aus Zustand und Signal die Übergangsfunktion maximal einen neuen möglichen Zustand festlegt
- die FSM ist „**finite**“ = endlich, da sie nur eine endliche Menge von Zuständen besitzt



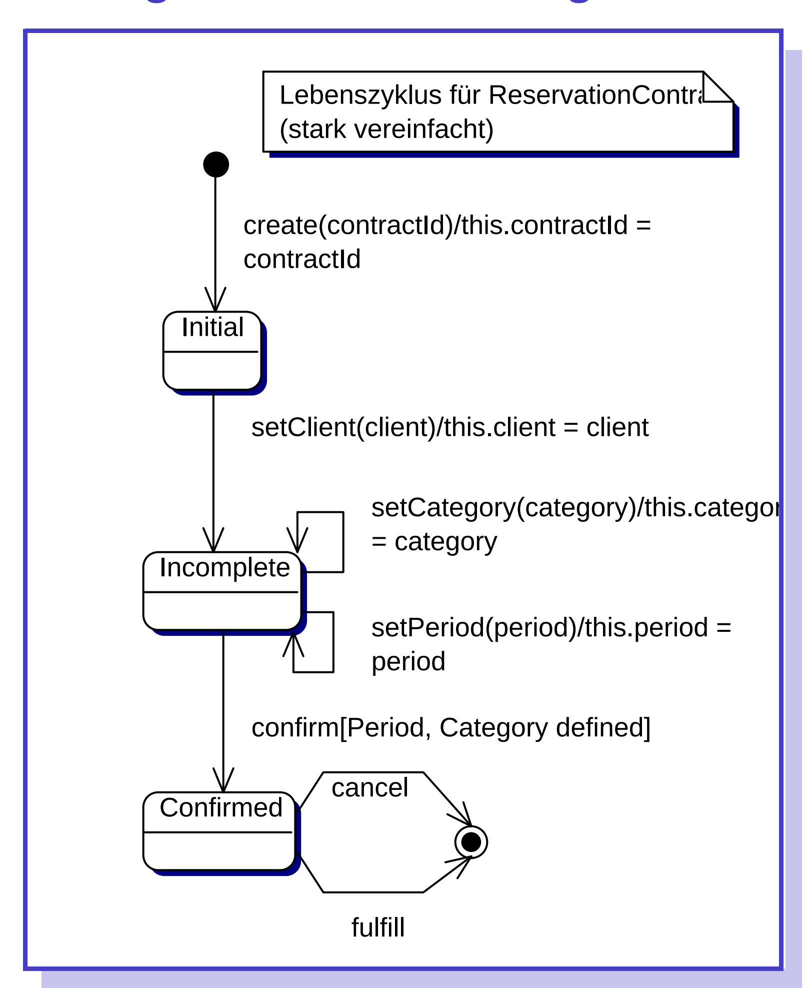
Deutungen von Zustandsdiagramm als Lebenszyklus:



- Diagramm beschreibt, in welcher Reihenfolge die Operationen der Klasse ReservationContract aufgerufen werden dürfen
- das Diagramm ist keine Implementierung der Klasse
- man spricht von **Lebenszyklus** oder **Protokoll** der Klasse



Deutung von Zustandsdiagramm als Klassenimplementierung:



signal [condition] / action

Bedeutung der Transitionsbeschriftung:

1. wenn signal eintrifft schaltet Transition
2. falls condition über Attributen erfüllt ist
3. und führt dabei ununterbrechbare action aus (die Attributwerte ändern können)

Merke:

Viele CASE-Tools können aus solchen Zustandsdiagrammen guten Code generieren.

Alle Teile der Aufschrift sind optional:

1. Transition ohne Aufschrift feuert sofort (sobald Zustand erreicht)
2. Transition nur mit Bedingung feuert, sobald Bedingung erfüllt ist
3. Transition mit Signal feuert, sobald Signal eintrifft
4. Transition mit Signal und Bedingung feuert, sobald Signal eintrifft, falls Bed. dann erfüllt



Statecharts = erweiterte Zustandsdiagramme [Ha87]:

- ☹ **Problem**: Zustandsdiagramme werden in realistischen Anwendungen groß!!!
- ☺ **Lösung**: Hierarchisierung der Zustandsdiagramme

UML State Machines ≈ Statecharts von Harel:

- erweiterte Zustandsdiagramme mit
  - ⇨ zusammengesetzten Zuständen
  - ⇨ „nebenläufigen" Teildiagrammen
- ein Diagramm wird immer genau einer Klasse zugeordnet
- jede Klasse besitzt höchstens ein Diagramm
- sie werden oft zur „Implementierung" von Objektverhalten benutzt
  (werden interpretativ ausgeführt oder in Quellcode übersetzt)



Statechart mit zusammengesetztem Zustand:

Transition in Anfangs-zustand

create

Initial

clear

setClient

Transition aus allen Unterzu-ständen

fulfill

cancel

Confirmed

Transition aus Endzustand

Incomplete

xor-state

WithPeriod

WithClient

Complete

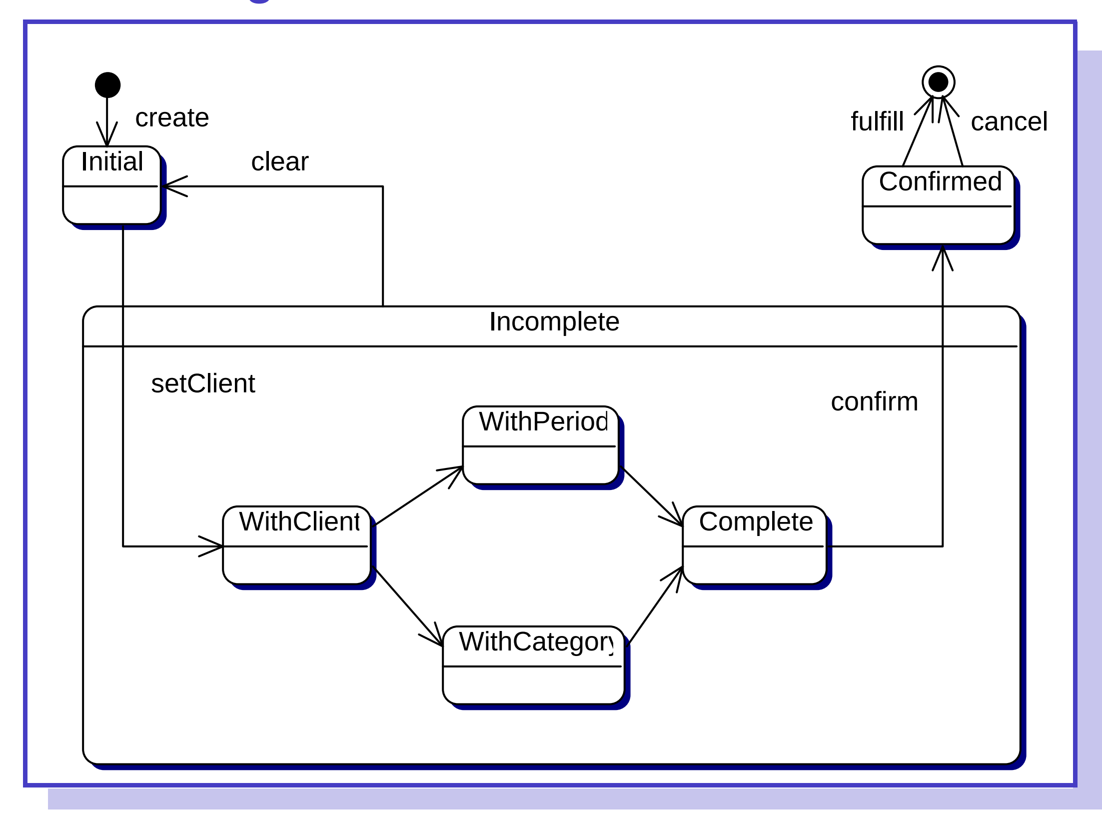
confirm

WithCategory

Lebenszyklus eines ReservationContract-Objektes mit **Oberzustand** Incomplete, der **Unterzustände** besitzt. Immer genau einer der Unterzustände von Incomplete ist aktiv (deshalb wird der Zustand xor-Zustand genannt).



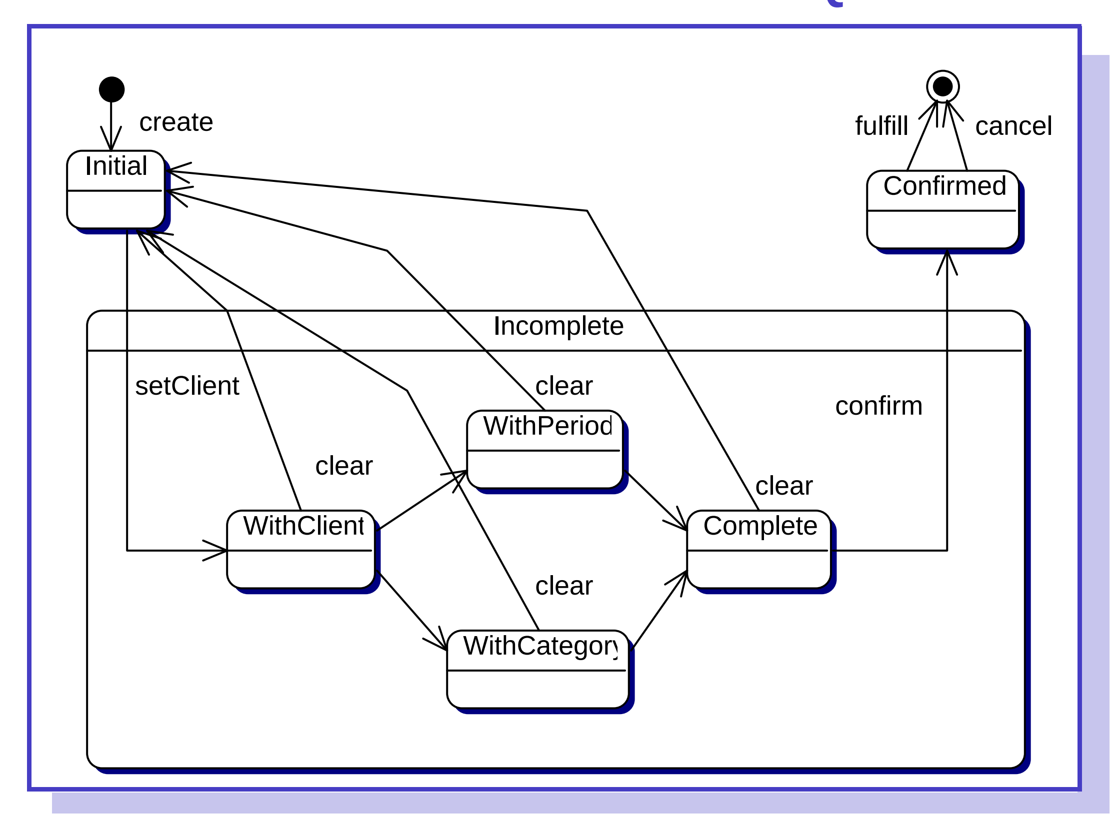
Elimination von Anfangs- und Endzuständen von xor-Zustand:



Die Transition in den Anfangszustand wurde auf den ersten „richtigen“ Zustand umgelenkt, die Transition aus dem Endzustand geht nun vom letzten „richtigen“ Zustand aus (hat die exakt dieselbe Bedeutung).



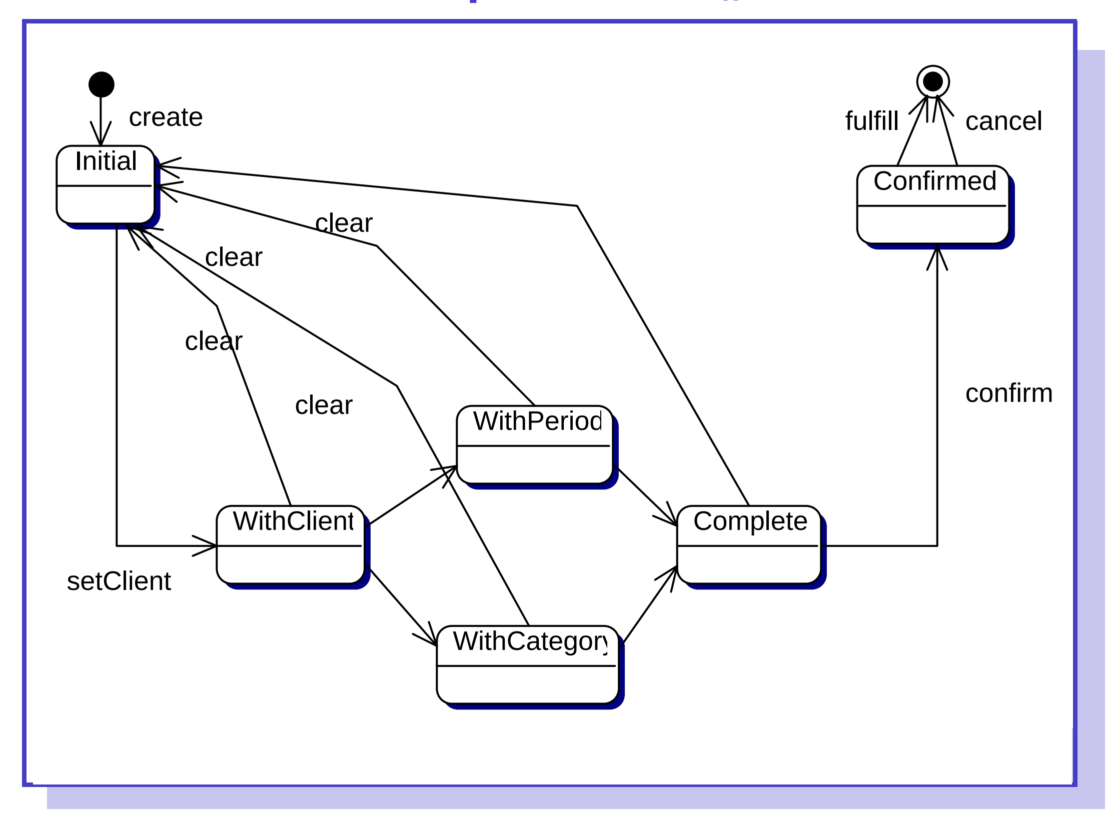
Elimination von Transition mit xor-Zustand als Quelle:



Die Transition mit Ereignis clear (alle Datenfelder wieder löschen) wurde durch eine entsprechende Transition von jedem Unterzustand aus ersetzt. Jetzt kann man den Oberzustand Incomplete entfernen.



Zu hierarchischem Statechart äquivalenter „flacher“ Automat:



Achtung: jedes „normale“ hierarchische Statechart lässt sich in einen normalen (flachen) Automaten übersetzen (Ausnahmen kommen später).



Abstraktes Beispiel mit xor-Oberzustand:

s1
s2
e1
e2
e3
e4
s3
s3.1
s3.2
e5
e6

Übersetzung in normales
Zustandsdiagramm

s1
s2
e1
e2
e3
e3
e4
s3.1
s3.2
e5
e6
e6



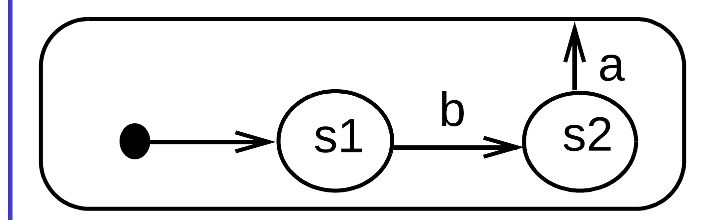
Übersetzung von Statecharts in einfache Zustandsdiagramme:

Statechart ⇔ **Zustands-diagramm**



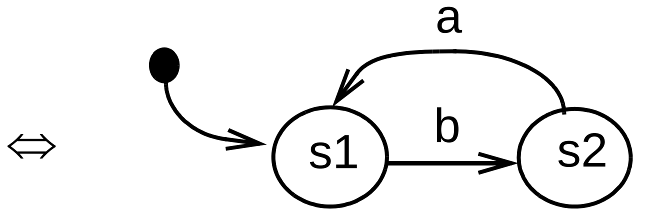
Übersetzung von Statecharts - 2:

Statechart

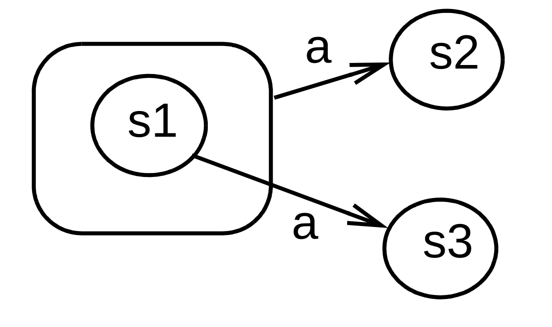


⇔

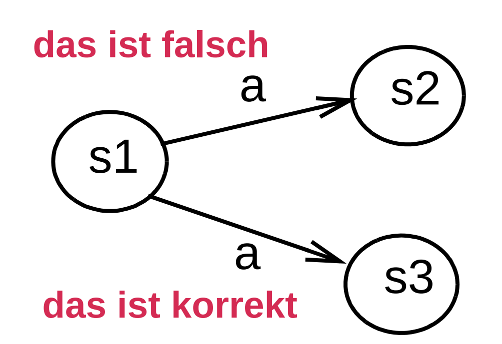
Zustands-diagramm



Problem:



?





And-Zustände (Concurrent States):

ReservationHandling

Incomplete

NoPeriod → enterPeriod → DefPeriod → ok

NoCategory → enterCat → DefCategory → ok

NoClient → enterClient → DefClient → ok

NoDeposit → enterDeposit → DefDeposit → ok

abort

Complete

paralleler Unterzustand

sobald abort-Signal kommt, wird In-complete in jedem Fall verlassen

diese Transition feuert, sobald alle Unterzustände ihren Endzustand erreicht haben

Anfangs-zustände

Signal ok führt zum Verlassen von Incomplete, falls alle Def-Zustände aktiv sind.

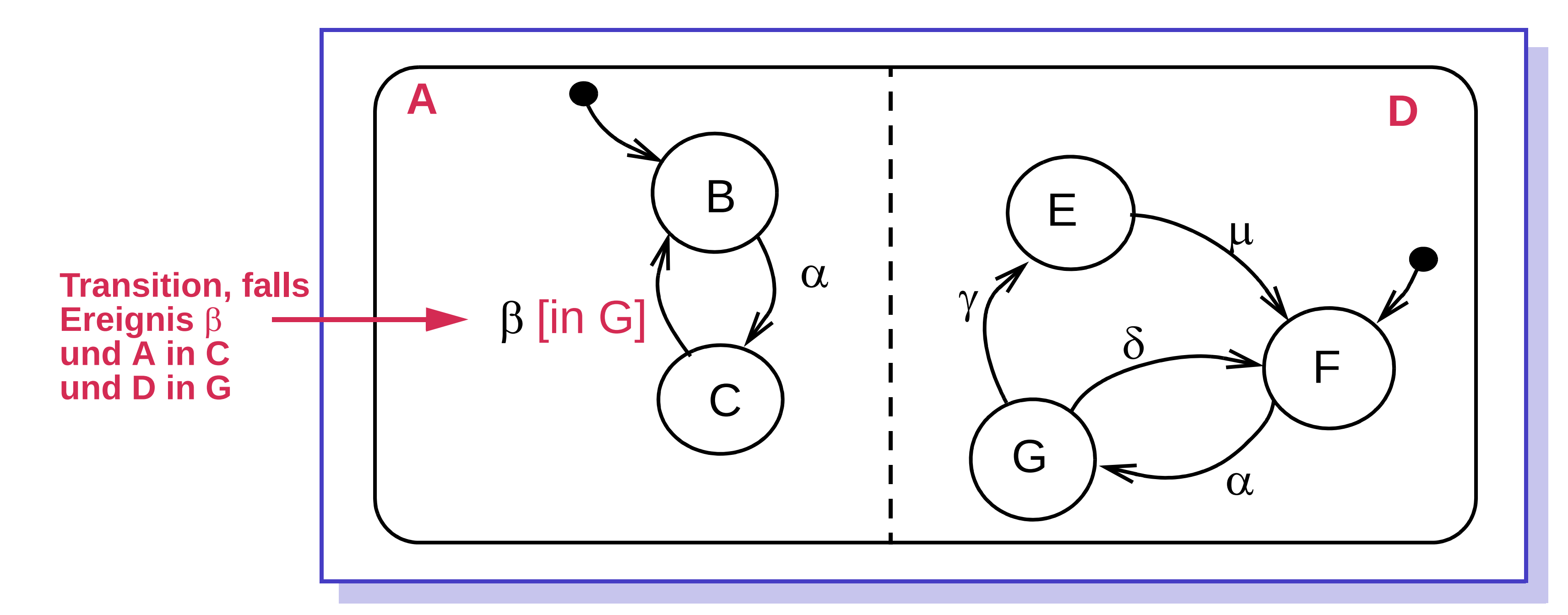


Erläuterung von and-Zuständen:

- ein and-Zustand enthält beliebig viele **Unterautomaten**
- beim Betreten des and-Zustands geht jeder Unterautomat in seinen **Anfangszustand** über
- jeder Unterautomat ist für sich aktiv (wie ein normaler Automat) und kann seinerseits wieder xor- und and-Zustände enthalten
- kommt ein **Signal** an, so wird dieses Signal von jedem Unterautomaten abgearbeitet (theoretisch gleichzeitig, in Praxis in **beliebiger** Reihenfolge)
- kann ein Unterautomat mit einem Signal nichts anfangen, bleibt er einfach im aktuellen Zustand
- ein Unterautomat wird entweder durch Transition nach „aussen" mit explizitem Ereignis verlassen oder wenn **alle** Unterautomaten ihre **Endzustände** erreicht haben (**aber**: alte UML-Büchern und -Werkzeuge behaupten ggf. anderes)
- Es gibt wenige UML-CASE-Tools, die **Ausführung** von Statecharts mit **and-Zuständen** unterstützen (ohne Unterstützung von and-Zustände gibt es viele)



Abstraktes Beispiel mit and-Oberzustand:



Achtung:

- die Unterautomaten A und D schalten „nebenläufig“ (bei passendem Ereignis)
- Querbezüge auf Zustand des „Nachbarautomaten” möglich mit „[in X]” als Transitionsbedingung (aber kein guter Modellierungsstil)



Übersetzung von and-Oberzustand in Produktautomaten:

B,F
B,E
B,G
C,E
C,G
C,F
μ
δ
α
γ
β



„Kochrezept“ für die Übersetzung von And-Oberzustand in Produktautomat:

1. Es wird das Kreuzprodukt der Zustände aller Regionen des And-Oberzustandes gebildet (also alle möglichen Kombinationen der Unterzustände)

2. Für jeden neuen Zustand (z_1, ... , z_n) und alle möglichen Signale s wird überprüft, ob es genau eine Transition mit Aufschrift s [b] / a von z_i nach z_i’ gibt.

3. Wurde in Schritt 2 eine solche Transition gefunden, dann wird neue Transition mit Aufschrift s [b] / a von (z_1, ... , z_i, ... , z_n) nach (z_1, ... , z_i’, ... , z_n) eingeführt

4. Wenn in Schritt 2 mehrere solche Transitionen gefunden wurden, dann
   - ⇨ werden mehrere Urzustände im Tupel (z_1, ... , z_i, ... , z_n) ausgetauscht
   - ⇨ werden für alle mögliche Kombinationen der Bedingungen b_i neue Transitionen angelegt
   - ⇨ werden die Aktionen ai der ursprünglichen Transitionen mit wahren Bedingungen b_i in irgendeiner Reihenfolge als Gesamtaktion ausgeführt

5. Zum Abschluss werden nicht erreichbare Kreuzprodukt-Zustände eliminiert



Regeln für die Gestaltung von Statecharts:

- Objekteigenschaften mit größerem Wertebereich nicht in Zuständen halten (Alter eines Autos, Gehaltsstufe einer Person, …)
- Teildiagramme mit mehr als 5 bis 7 Zuständen vermeiden (durch Einführung von or- und and-Zuständen)
- nie Zustände verwenden, die Informationen zu mehreren Eigenschaften repräsentieren (z.B. Zustand “Auto ausgeliehen und Inspektion fällig”)
- Kopplung der Teildiagramme eines and-Zustandes über [in State]-Bedingungen (Wächter) nur überlegt einsetzen (führt oft zu schwer verständlichen Diagrammen)
- anstelle eines Objektes mit and-Zustand oft lieber mehrere Objekte mit einfachen Statecharts verwenden (die über Schnittstellenoperationen kommunizieren)
- an die Verwendung ereignisloser Transitionen denken, die bei Erfülltsein einer Bedingung schalten (z.B. Kilometerstand eines Autos größer als …)



Von sequentiellen Zustandsdiagrammen zu parallelen Statecharts:

- Zustandsdiagramme beschreiben **sequentielle** Abläufe in einem Objekt:
  - ⇨ zu einem Zeitpunkt löst ein Ereignis maximal einen Zustandsübergang aus
  - ⇨ ein Event löst maximal eine Aktion aus
- and-Zustände “normaler” Statecharts beschreiben (fast) **nebenläufige** Abläufe:
  - ⇨ zu einem Zeitpunkt löst ein Ereignis maximal eine Transition im Produktautomaten aus (und damit gleichzeitig Menge von Teiltransitionen)
  - ⇨ Aktionen der Teiltransitionen werden in beliebiger Reihenfolge ausgeführt

Aber:

- UML-Statecharts erlauben auch echt **parallele** Abläufe (mit fork/join):
  - ⇨ während Aktionsausführung zu einem Ereignis in Teilautomat A können bereits neue Ereignisse in anderem Teilautomat B behandelt werden
  - ⇨ jeder Teilautomat hat eigene (Kopie der) Warteschlange mit eingetroffenen Ereignissen (die noch abzuarbeiten sind)



Komplexe = echt parallele Zustandsübergänge:

Achtung:

Die and-Zustände mit fork/join-Konstrukt lassen sich nicht mehr in Produktautomaten übersetzen. Man hat in einem Objekt mehrere „**threads of control**”.



Beispiel - Lebenszyklus von ReservationContract-Objekten:

incomplete
setClient
new
setPeriod
set Category
setDeposit
setVehicle Data
ok

- Zustandsdiagramme besitzen Ähnlichkeit mit Aktivitätsdiagrammen; Knoten sind aber hier **Zustände genau eines Objekts** und keine Aktionen

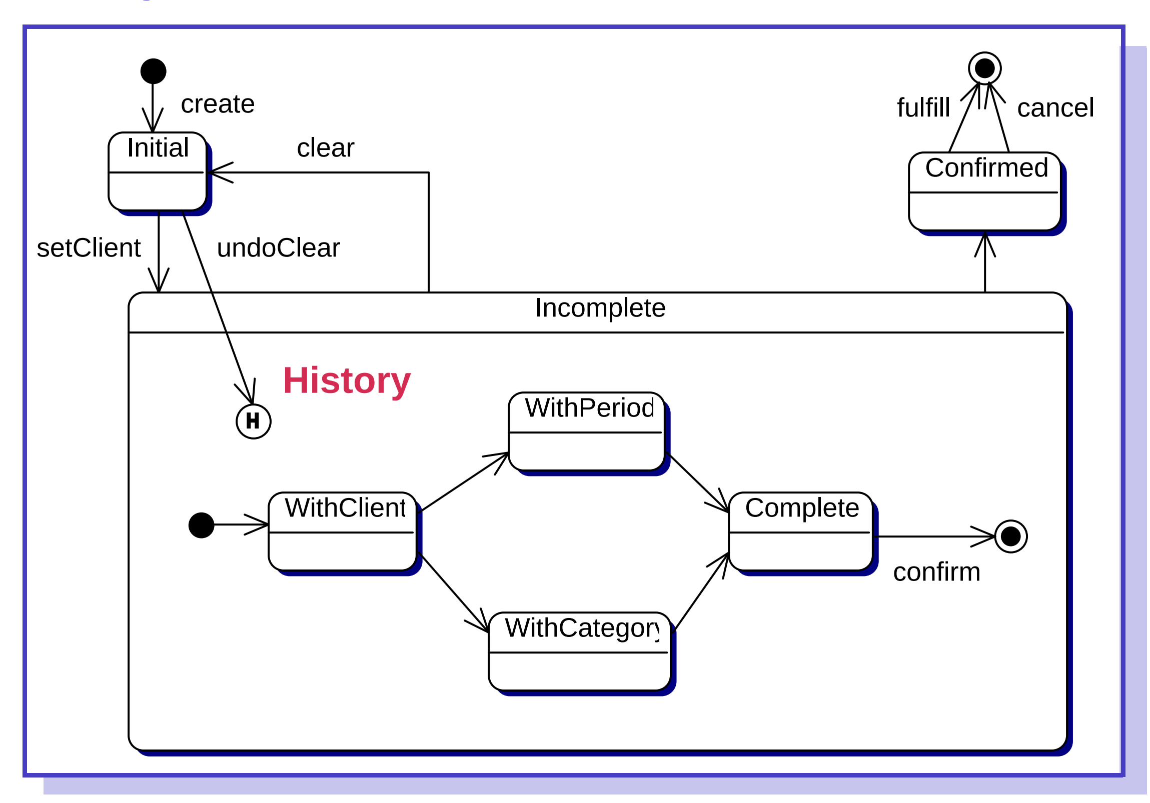


Weitere Elemente von UML-Statecharts:

- **History-Mechanismus** in komplexen Zuständen:
  bei Wiedereintritt in Zustand wird nicht Anfangszustand aktiviert, sondern der letzte Unterzustand aus dem Diagramm beim letzten Mal verlassen wurde
  (für Ausnahmebehandlungen)
- **Eintritts und Austrittsaktionen** (entry/exit actions)von Zuständen:
  nicht nur Transitionen sind mit Aktionen verbunden, sondern auch Eintritt in oder Austritt aus Zustand kann Aktionen auslösen
- **Eintritts- und Austrittspunkte** (entry/exit points) komplexer Zustände:
  will man Unterzustände an mehreren Stellen wiederverwenden, so müssen diese Schnittstellen besitzen, über die sie betreten und verlassen werden
- **Terminierungsknoten** (terminate): erreicht man diesen Knoten, so wird das Objekt, zu dem das Statechart gehört sofort gelöscht (und nicht erst dann, wenn Endezustand ganz außen erreicht wurde)
- ...



Beispiel für History-Mechanismus:



Wenn man mit undoClear den Zustand Incomplete **wieder** betritt, landet man in dem Unterzustand, von dem aus man mit clear den Zustand verlassen hatte, sonst im Anfangzustand WithClient.



Statechart mit den weiteren Sprachelementen:

StateMachine

Initial
setClient
undoCancel
Incomplete
ok
Confirmed
done
cancel

Entry-Point-Referenz

Exit-Point-Referenz

Exit-Point

Objektzerstörung (Terminate)

Verfeinerung

Entry-Point

Incomplete

enterClient
NormalEntry
WithClient
enterPeriod
WithPeriod
enterCategory
WithCategory
enterCategory
enterPeriod
Complete
kill
ok
NormalExit
undoCancel
Reentry
H
CancelExit



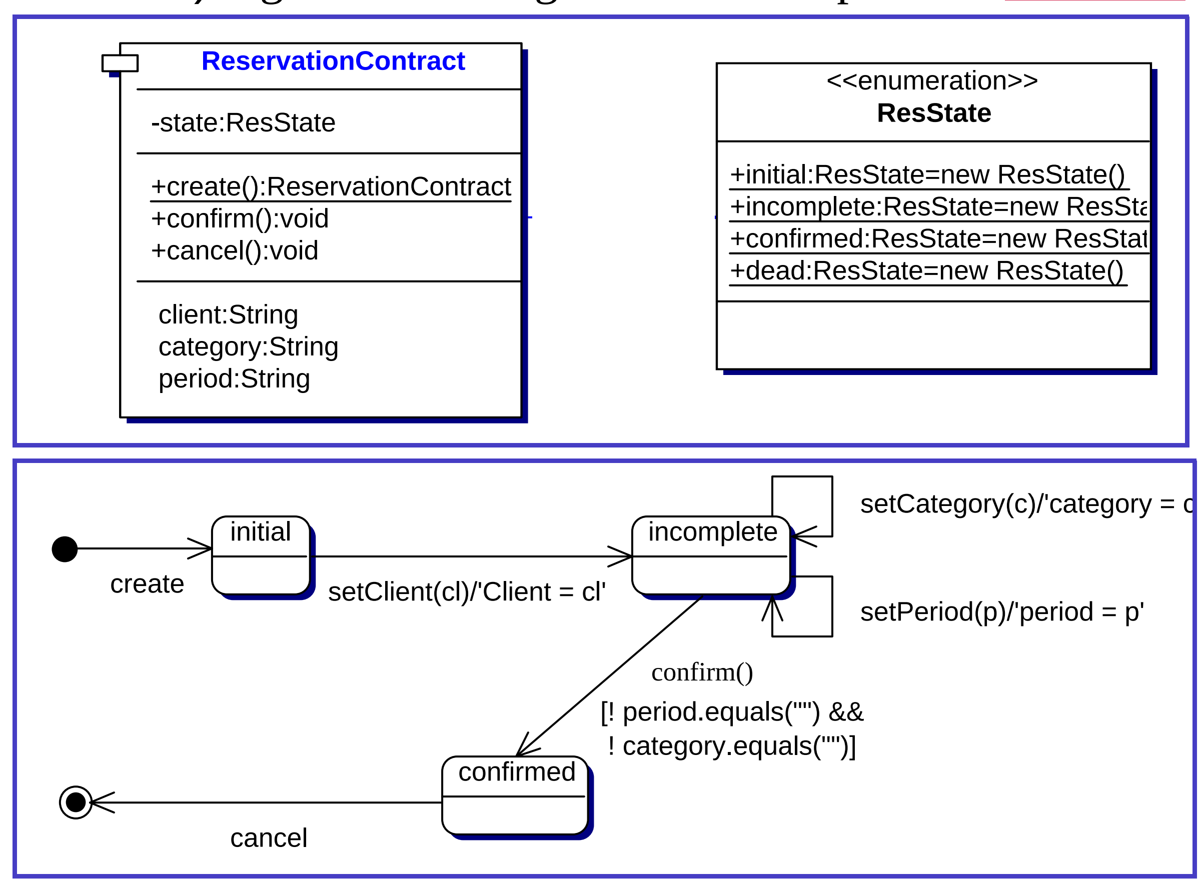
Erläuterungen zu dem neuen Beispiel:

- der separat definierte XOr-Zustand Incomplete könnte in mehreren Statechart-Diagrammen an verschiedenen Stellen eingesetzt (referenziert) werden
- er besitzt zwei Eintritts- und zwei Austrittspunkte, die bei der Verwendung durch Transitionen referenziert werden; dadurch kann man Transitionen von außen zu bestimmten internen Zuständen führen ohne diese zu kennen
- Incomplete muss beim ersten Mal über NormalEntry betreten werden
- verlässt man Incomplete über NormalExit (aus Zustand Complete) oder über CancelExit (aus beliebigem Zustand), wird der vorher erreichte Zustand vermerkt (als History-State)
- betritt man den Zustand Incomplete wieder über Reentry, so landet man in dem Zustand, in dem man beim letzten Verlassen von Incomplete am Ende gewesen war
- die Transition kill von Complete aus löscht sofort das zugehörige Objekt



Codegenerierung aus Statecharts:

Mit Codefragmenten und Definition des Aufzählungstyps ResState (für ReservationContract-Zustände) ergänztes leicht geändertes Beispiel von Seite 389





Generierbarer Code aus Statechart:

```
public class ReservationContract {
   public String getClient(){ return client; }
   public void setClient(String client){
      if (state == ResState.initial){
         this.client = client;
         state = ResState.incomplete;
      }
      // else tue nichts, man könnte höchstens eine Ausnahme auslösen
   }
   …
   public void confirm() {
      if (state == ResState.incomplete && ! category.equals("") && ! period.equals(""))
         state = ResState.confirmed;
      // else tue nichts, man könnte höchstens eine Ausnahme auslösen
   }
   private String client = "";
      …
   private ResState state = ResState.initial;
}
```



7.5 Zusammenfassung





Konsistenzbedingungen zwischen Diagrammen:

- **Klassendiagramm ↔ Objektdiagramm**:
  jedes Objektdiagramm ist zulässige Instanz des zugehörigen Klassendiagramme
- **Klassendiagramm ↔ Sequenzdiagramm**:
  jede Operation muss bei der richtigen Klasse (richtig) deklariert sein
- **Sequenzdiagramm ↔ Zustandsdiagramm (Statechart)**:
  Operationsfolgen des Sequenzdiagramms sind zulässig im Zustandsdiagramm; Ausführungen von Zustandsdiagrammen sollten alle beschriebenen Abläufe von Sequenzdiagrammen erzeugen können
- **Klassendiagramm ↔ Zustandsdiagramm (Statechart)**:
  jede Klasse besitzt maximal ein eigenes Implementierungs-Zustandsdiagramm (plus ggf. Lebenszyklus-Zustandsdiagramm zu Schnittstellen); Ereignisse sind Operationen der Klasse (wie Vererbung von Zustandsdiagrammen handhaben?)
- **Aktivitätsdiagramm ↔ ???**: noch immer etwas unklar



7.6 Weitere Literatur

[BC89] Alhir S.: *UML in a Nutshell - A Desktop Quick Reference*, O'Reilly (1998)

Hält leider nicht das was es verspricht! Preiswerte und relativ vollständige Präsentation von UML, aber sowohl als Einführung als auch als Nachschlagewert nicht sonderlich gut geeignet.

[BC89] K. Beck, W. Cunningham: *A Laboratory For Teaching Object-Oriented Thinking*, in: Proc. OOPSLA'89, SIGPLAN Notices, Vol. 24, No. 10, ACM Press (1989), 1-6

Die Originalquelle zum Thema CRC-Karten (die für die Identifikation von Klassen benutzt werden).

[BD00] B. Bruegge, A.H. Dutoit: *Object-Oriented Software Engineering*, Prentice Hall (2000), 553 Seiten

Basiert auf den Erfahrungen mit der Durchführung von Praktika zur objektorientierten Softwareentwicklung an der TU München und der Carnegie Mellon University. Beschreibt eine ganze Reihe von Faustregeln für Projektmanagement, Anforderungsanalyse, Erstellung von UML-Diagrammen, … . Es ist schade, dass die verwendeten Beispiele dauernd wechseln.

[Ha87] Harel, D.: *Statecharts: A visual formalism for complex systems*, Science of Computer Programming, vol. 8, Elsevier Science Publ. (1987), 231-274

Originalliteratur, in der hierarchische Automaten als sogenannte Statecharts erfunden wurden.

[La98] Larman C.: *Applying UML and Patterns*, Prentice Hall (1998)

Eines der ersten UML-Bücher, das anhand eines durchgängigen Beispiels ein Vorgehensmodell zum Einsatz von UML vorstellt. Eine vereinfachte und abgeänderte Version dieses Vorgehensmodells wird in dieser Vorlesung benutzt.



8. Objektorientierter Softwareentwurf

Themen dieses Kapitels:

- endgültiger Übergang von der Analyse zum Entwurf
- Modellierung von Softwarearchitekturen mit weiteren UML-Diagrammarten
- Erkennung und Restrukturierung (Refactoring) „schlechter“ Software
- Einsatz von Entwurfsmustern (Design Pattern) als Standardlösungen

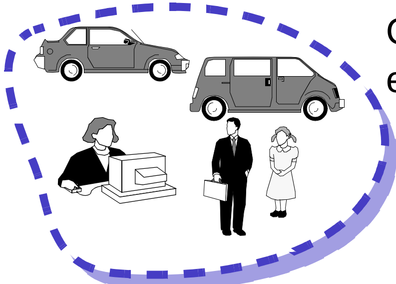
Achtung:

- Ergebnis der Entwurfsphase ist ein detaillierter **Bauplan** des Softwareprodukts mit Festlegung von Verantwortlichkeiten für Realisierung, Test, … von Teilsystemen
- Ausgangspunkt ist das in der Analyse ermittelte **Fachkonzept**, die Produktdaten und -funktionen des zu realisierenden Systems



8.1 Von der Analyse zum Entwurf

Problemgebiet



Objekte und Abläufe im Umfeld einer Autovermietung

Analysemodell

Produktfunktionen und -daten aus Anwendersicht

Entwurfsmodell

Produktfunktionen und -daten aus Entwicklersicht

→ **Architektur des Produkts**

Implementierung des Motor Vehicle Reservation Systems

Programm



Typische Bestandteile einer Softwarearchitektur:

- anwendungsspezifische Funktionen und Daten (**in Analyse betrachtet**)
- Details der Benutzungsoberfläche (Standards, GUI-Bibliotheken, …)
- Ablaufsteuerung (mit Transaktionen, …)
- Datenhaltung (in Dateien, Datenbanken, …)
- Infrastrukturdienste für
  - ⇨ Objektverwaltung (Garbage-Collection, …)
  - ⇨ Prozesskommunikation
- Sicherheitsfunktionen (Verschlüsselung, Passwortschutz, …)
- Zuverlässigkeitsfunktionen (Fehlererkennung und -behebung, …)
- Systemadministration (Statistiken, Installation, Sicherungen, …)
- weitere Basisbibliotheken (für arithmetische Funktionen, …)



Zur Erinnerung:

Planung

Kundenanforderungen
(z.B. als Lastenheft)
(siehe **Abschnitt 3.3**)

Anwendungsfälle
→ UML Use Cases
(siehe **Abschnitt 5.2**)

?

"Workflow"-Modell
→ UML Activity Diagrams
(siehe **Abschnitt 5.5**)

Analyse

konzept. Datenmodell
→ UML Class Diagrams
(siehe **Abschnitt 5.3**)

?

feines Verhaltensmodell
→ UML Interaction Diag.
(siehe **Abschnitt 6.2**)

feines Klassenmodell
→ Class Diagram mit Ops.
(siehe **Abschnitt 6.1**)

?

reaktives Verhalten
→ UML State Machines
(siehe **Abschnitt 6.3**)

(Analyse →) Design



Bereits bekannte Architektursichten auf ein Softwareprodukt:

- **„klassische“ Struktur-Sicht**: Klassenhierarchien mit Beziehungen, …
  - ⇨ mit Klassendiagrammen (hauptsächlich hier verwendete Architektursicht)
- **Datenfluss-Sicht**: wie fließen Daten von Aktion zu Aktion durch das System
  - ⇨ mit Aktivitätsdiagrammen
- **Kontrollfluss-Sicht**: welche Operationen (Klassen) rufen sich gegenseitig in welcher Reihenfolge auf
  - ⇨ mit Aktivitäts- und Sequenzdiagramme
- **Prozess-Sicht**: Beschreibung dynamischer Aspekte mit Festlegung sequentieller und (potentiell) paralleler Abläufe, Kommunikationsarten etc.
  - ⇨ mit Sequenzdiagrammen (Statecharts)

Merke:

> **Architektur eines Softwareprodukts ist mehr als Zerlegung in Teilsysteme**



Neue Architektursichten auf ein Softwareprodukt:

- **Teilsystem-Sicht**: Zerlegung des Quellcodes in unabhängig bearbeitbare Teile mit präzise definierten Schnittstellen
  - ⇨ in UML mit Paketdiagrammen (Package Diagrams) (siehe Abschnitt 8.2)
- **Konfigurations-Sicht**: Zusammenstellung ausführbarer Systemversionen als ausführbare Dateien mit zusätzlichen Tabellen, Online-Manualen, …
  - ⇨ in UML mit Verteilungsdiagrammen (Deployment Diagrams) (siehe Abschnitt 8.3)
- **Physikalische Sicht**: Abbildung von Software auf Hardware (oder Betriebs-systemdienste), insbesondere bei verteilten und/oder eingebetteten Systemen
  - ⇨ in UML mit Verteilungsdiagrammen (Deployment Diagrams) (siehe Abschnitt 8.3)


Fachgebiet
Echtzeit-
systeme


Weitere UML-Diagrammarten:

Analyse
Kundenanforderungen
(z.B. als Lastenheft)
Anwendungsfälle
→ Use Cases
"Workflow"-Modell
→ Activity Diagrams
konzept. Datenmodell
→ Class Diagrams
feines Verhaltensmodell
→ Interaction Diagrams
feines Klassenmodell
→ Class Diagrams
reaktives Verhalten
→ Statecharts
?
?
?
Teilsystemmodellierung
→ Package Diagrams
(siehe Abschnitt 8.2)
Softwarekonfigurationen
→ Deployment Diagrams
(siehe Abschnitt 8.3)
Abbildung auf Hardware
→ Deployment Diagrams
(siehe Abschnitt 8.3)
Design



Neue Diagrammarten in UML (für die Entwurfsphase):

- **Paketdiagramme** zur Modularisierung (wie in Java, ...):
  - ⇨ als hierarchisches Dateisystem für Programmteile
  - ⇨ zur Bildung von Bibliotheken/Teilsysteme
- **Verteilungsdiagramme (Installationsdiagramme)**: Komponenten → Hardware
  - ⇨ fasssen Klassen und Konfigurationsbeschreibungen zu verteilbaren und installierbaren Dateien/Archiven/... zusammen
  - ⇨ beschreiben Struktur der Hardware (Bestandteile, Verbindungen, …) (oder Struktur von Betriebssystemprozessen)
  - ⇨ bestehen aus verteilbaren (Software-)Artefakten sowie aus (Hardware-)Knoten und Verbindungen zwischen Knoten
  - ⇨ ordnen Artefakte bestimmten Knoten zu



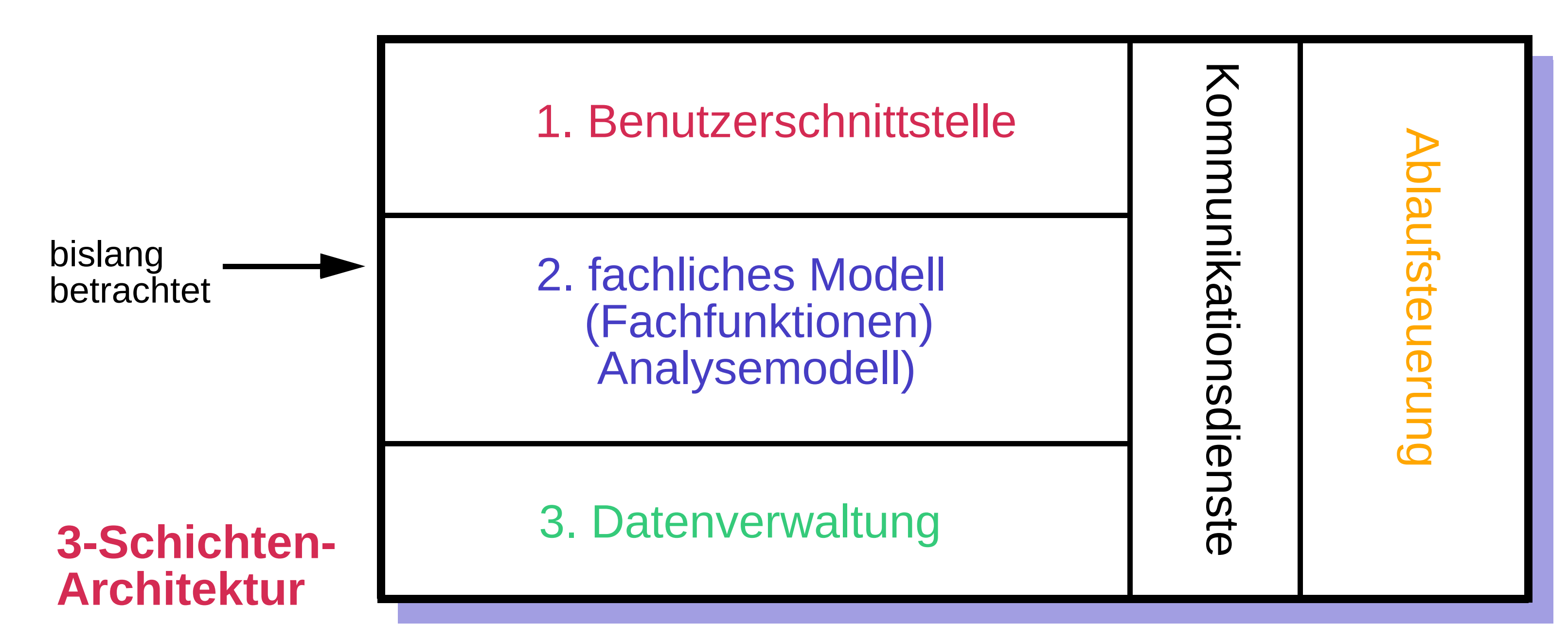
8.2 Modularer Softwareentwurf mit der UML (Package Diagrams)

Klassen- und Paketdiagramme bilden in UML die Basis für die Zerlegung eines Softwaresystems in Teilsysteme (Module) und Teilsystemen von Teilsystemen etc.:

- **Klassen** = atomare Teilsysteme:
  - ⇨ Bestandteile sind Attribute und Operationen
  - ⇨ Bestandteile haben Sichtbarkeiten (public, protected, private)
- **Pakete in der Analyse** = komplexe Teilsysteme:
  - ⇨ Bestandteile sind Unterpakete oder zusammengehörige Diagramme
  - ⇨ realisieren einfach nur hierarchisches Dateisystem
  - ⇨ Sichtbarkeiten spielen kaum eine Rolle
- **Pakete beim Entwurf** = komplexe Teilsysteme:
  - ⇨ wie in der Analyse benutzte Pakete
  - ⇨ Sichtbarkeiten sind aber sehr wichtig



Standard-Bauplan eines interaktiven Softwaresystems:



Prinzipien für die Zerlegung in Teilsysteme:

- drei Hauptschichten: Benutzerschnittstelle, Fachfunktionen, Datenhaltung
- Ablaufsteuerung regelt Zusammenspiel Benutzerschnittstelle und Fachfunktionen
- Kommunikationsdienste erlauben Kommunikation zwischen Teilsystemen



Aufgaben der Teilsysteme am sogenannten „MVC“-Muster-Beispiel:

- Benutzerschnittstelle (**V**iew = Sicht auf Daten = „V“ aus „MVC“):
  - ⇨ alle Bestandteile des Systems, die mit Benutzeroberfläche zu tun haben
  - ⇨ Realisierung mit einer Benutzeroberflächenbibliothek (awt, Swing, …)
- Fachliches Modell (**M**odel = Logik = „M“ aus „MVC“):
  - ⇨ verfeinertes Ergebnis der Analysephase
  - ⇨ nutzt Datenverwaltung für die Speicherung von Fachdaten
- Ablaufsteuerung (Control = Kontrolle = „C“ aus „MVC“):
  - ⇨ regelt das Zusammenspiel zwischen „Model“ und „View“
  - ⇨ steuert Ausführungsreihenfolge von Funktionen
- Datenverwaltung (optionale Ergänzung des MVC-Musters):
  - ⇨ speichert alle Daten (in Dateien oder einer Datenbank)
- Kommunikationsdienste („Klebstoff“ für die MVC-Bestandteile):
  - ⇨ unterstützen Kommunikation über Prozess- oder Rechnergrenzen hinweg



Aufgaben eines Teilsystems = Paket = Modul im allgemeinen:

- dient einem klar umrissenen Zweck
  - ⇨ "**separation of concerns**"
- fasst eng zusammengehörige Dienste (Daten, Funktionen) zusammen
  - ⇨ "**high cohesion**" (hohe Kohäsion innerhalb eines Teilsystems)
- besitzt ein Geheimnis (verborgene Implementierungsdetails)
  - ⇨ "**information hiding**"
- benutzt wenige Dienste anderer Teilsysteme
  - ⇨ "**low coupling**" (lose Kopplung zwischen Teilsystemen)
- besteht aus **Schnittstelle** (angebotene Dienste) und **Rumpf** (ihre Realisierung)
- lässt sich als **eigenständige Einheit** entwickeln, übersetzen und testen
- bildet ein sinnvolles **Arbeitspaket** für eine Person oder Gruppe



Hauptnutzen von Teilsystemen = Moduln = Pakete:

- Lokalisierung vorhersehbarer Änderungen an einer Stelle
- Schaffung wiederverwendbarer Softwarebestandteile
- Voraussetzung für Parallelisierung von Entwicklungstätigkeiten

Beispiel - Paket mit Implementierung des Datentyps “Datum”:

- Umstellung von zwei- auf vierstellige Repräsentation von Jahreszahlen geschieht in genau einem Paket (statt an allen Programmstellen, die Datum bearbeiten)
- Datumsmodul lässt sich wiederverwenden (Fehler bei der Berechnung von Schalt-jahren etc. werden nicht in jedem Softwareprodukt neu gemacht)

Erwartung:

Modularisierung erhöht Produktivität und senkt Fehlerrate



Beispiele für typische Änderungen:

- Wechsel von **Algorithmen**:
  z.B. Sortieralgorithmus Quicksort → Mergesort
- Wechsel von **Datenstrukturen**: z.B. Mengenimplementierung mit linearer Liste → Bit-Array
- Wechsel von **Hardware**:
  z.B. neue Ein- und Ausgabegeräte, neue Rechner, …
- Wechsel von **„abstrakten Maschinen“**:
  z.B. neues Betriebssystem, Fenstersystem, andere Datenbank, …
- Wechsel der **Benutzeroberfläche**:
  z.B. Benutzeroberfläche für Großrechnerterminal → grafische Oberfläche auf PC
- Wechsel des **sozialen Umfelds**:
  z.B. Erhöhung der Mehrwertsteuer, Umstellung von DM auf Euro



Umsetzung der „Standard-Architektur“ in UML:

MVC = Model + View + Control



Standard-Stereotypen für Klassen - am Beispiel MVC-Konzept:

- (Benutzer-)Schnittstellenklassen erhalten den Stereotyp **<<boundary>>** (im Folgenden in allen Diagrammen meist rot dargestellt); u.a. alle im MVC-Konzept dem „**View**“ zugeordneten Klassen
- Klassen, die hauptsächlich der Datenhaltung dienen, erhalten den Stereotyp **<<entity>>** (im Folgenden meist blau dargestellt); u.a. alle im MVC-Konzept dem „**Model**“ zugeordnete Klassen
- Kontrollklassen, die die Interaktionen anderer Klassen steuern, erhalten den und selbst kaum Daten speichern, erhalten den Stereotyp **<<control>>** (im Folgenden meist gelb dargestellt); u.a. alle im MVC-Konzept der „**Control**“ zugeordneten Klassen

Selbst eingeführte Stereotypen:

Für bestimmte Anwendungsbereiche kann die Einführung weiterer Stereotypen sinnvoll sein, wie etwa <<sensor>> bei eingebetteten Systemen oder <<transaction>> und **<<database>>** bei Datenbankanwendungen oder … .



Dann Zerlegung des Klassenmodells der Analyse (Model):

MVRS
UIMS
Legal Entity
{disjoint}
Company
Person
{overlapping}
Rental Office
works For
Employee
Client
hasClients ▶
has Reservation Contracts ▶
has Rental Contracts ▶
Contract
{disjoint}
Reservation Contract
Rental Contract
Insurance Contract
hasInsurance ▶
hasBill
Bill
hasCategory
Category
isOfCategory ▶
◀ reservedVehicle
◀ rentedVehicle
hasVehicles ▶
Motor Vehicle
{overlapping}
Car
Truck
Van

Behandlung paketübergreifender Assoziationen?



Pakete für fachliches Modell:


```
package MVRS.Model.LegalEntities;
  import
     MVRS.Model.Vehicles.* ;
// oder nur Motorvehicle importieren:
// import … .Vehicles.Motorvehicle
   …
```



Achtung bei Assoziationen über Paketgrenzen hinweg:

Ist eine Assoziation gerichtet, dann muss nur Paket der Quellklasse die Zielklasse importieren, ansonsten benötigt man Import in beide Richtungen. Das sollte man - wenn möglich - vermeiden und Klassen in einem Paket deklarieren.

Beispiel mit ungerichteter „has“-Assoziation:

```
package MVRS.Model.LegalEntities;
   import MVRS.Model.Vehicles.Motorvehicle;
   public class RentalOffice extends Company {
      private Employee lnkEmployee;
      private Motorvehicle lnkMotorVehicle;
   … }
package MVRS.Model.Vehicles;
   import MVRS.Model.LegalEntities.RentalOffice;
   public class Motorvehicle {
      private RentalOffice lnkrevRentalOffice;    // der Rückwärtsverweis !!!
   … }
```



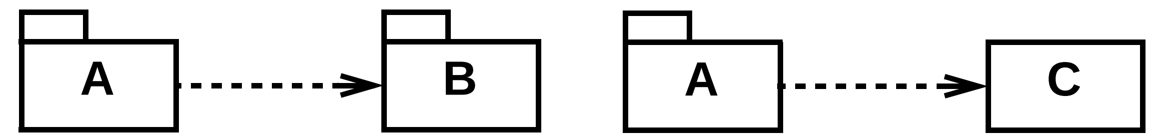
Eigenschaften von Paketen und Sichtbarkeiten:

- ein Paket U kann **Unterpaket** eines Pakets P sein und besitzt dann den Namen P::U (mit beliebiger Schachtelung)
- alle Inhalte eines Paketes (Unterpakete, Klassen, Assoziationen, …) besitzen **Sichtbarkeiten** (viele CASE-Tools und Programmiersprachen unterstützen das aber nicht im vollen Umfang)
- die Sichtbarkeiten wurden bereits eingeführt und sind:
  - „+“ = public (durch Import sichtbar)
  - „#“ = protected (durch Vererbung sichtbar, nur in Klassen sinnvoll)
  - „-“ = private (nur lokal sichtbar, manchmal auch local genannt)
  - „~“ = package (nur im umgebenden Paket sichtbar, nur in Klassen für Attribute und Methoden sinnvoll)
- ein Unterpaket sieht alles was seine umfassenden Pakete sehen; Sichtbarkeit von Elementen wird also in Kindpakete vererbt



Schnittstellen und Beziehungen zwischen Klassen und Paketen:

- **Import-Beziehung** (Dependency):



⇨ Paket A importiert Paket B und sieht damit **alle** public-Elemente von B (in B enthaltene Klassen, Pakete, ...) = Export von B

⇨ Paket A importiert nur eine Klasse und sieht damit genau diese Klasse

- **Public und Private-Import**-Beziehung zwischen Paketen:



⇨ bei „normalem“ Import werden alle aus B importierten Elemente zu öffentlich sichtbaren (public) Elementen von A (und werden von dort weiter exportiert)

⇨ bei „access“-Import werden alle öffentlich sichtbaren Elemente von B zu nur lokal sichtbaren Elementen von A (werden nicht weiter exportiert)



8.3 Verteilungsdiagramme von UML

Die bisher eingeführten UML-Diagramme beschreiben das Softwareprodukt aus logischer Sicht. Bislang nicht betrachtet wurde

- ⇨ Zusammenfassung (des Codes) von Klassen und Verhaltensdiagrammen zu Binärdateien, die ausgeliefert werden
- ⇨ Zuordnung von Hilfsdateien zu einzelnen Binärdateien wie Datenbanken, Online-Hilfetexte, Fehlermeldungsdateien, Konfigurationstabellen, …
- ⇨ Verteilung von Softwareprodukt auf mehrere Hardwarekomponenten (oder Betriebssystemprozesse)

Vorgehensweise:

- ⇨ Beschreibung von physikalischen (verteilbaren) Softwarekomponenten und ihrer Abhängigkeiten mit **Artefakten**
- ⇨ Beschreibung der Zielumgebung (target system) und der Verteilung von Komponenten auf Zielumgebung mit **Knoten**



Die „Standard-Architektur“ zur Erinnerung:

MotorVehicleReservationSystem

MVRS

Platform

Unterpaket von MVRS

Control

Ablaufsteuerung

Control importiert View

Model
Contracts
LegalEntities
Vehicles

View

Benutzerschnittstelle

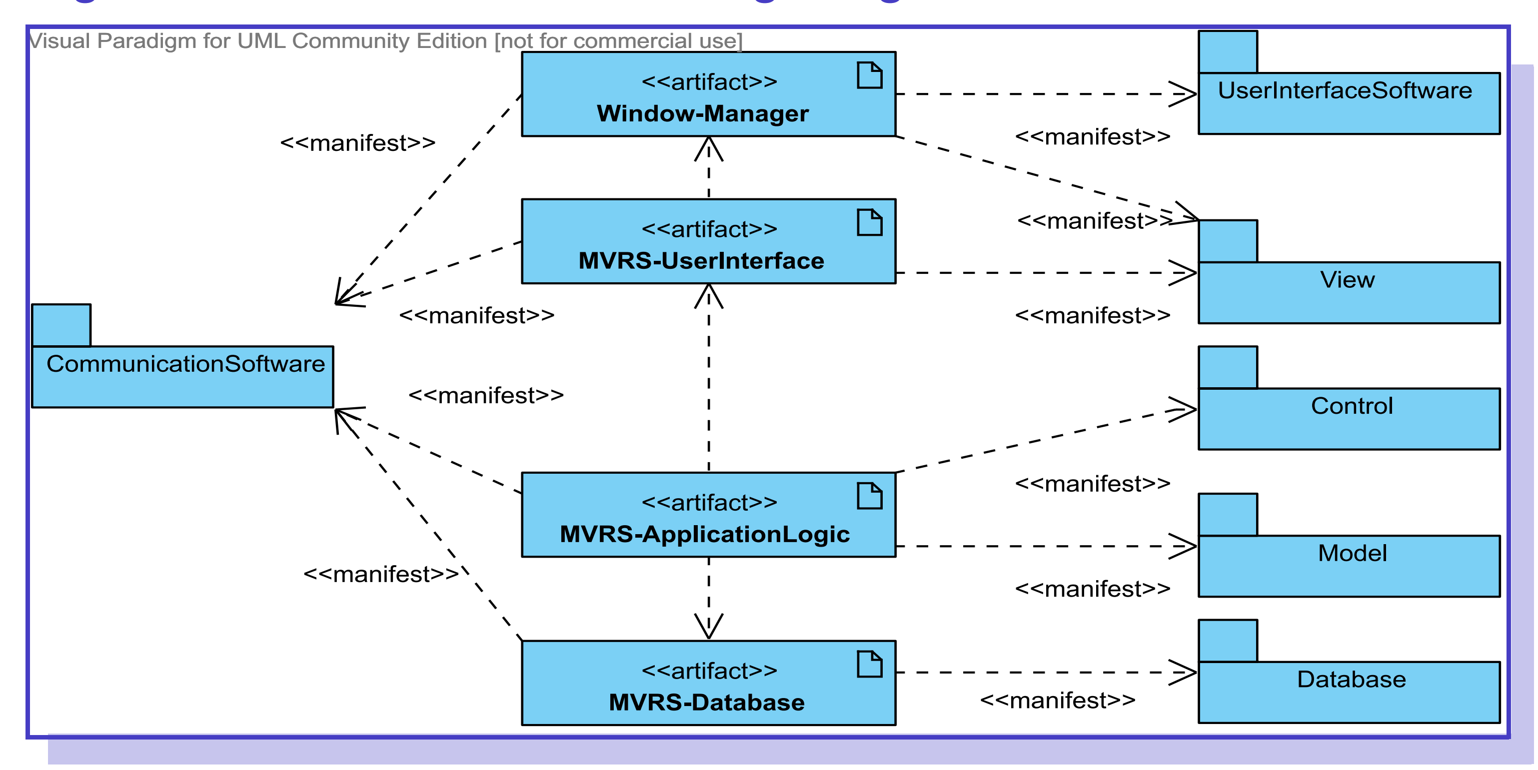
...DatabaseManagementSyste

...UserInterfaceSoftware

Platform.CommunicationSoftwa



Bildung von Artefakten mit Verteilungsdiagramm:



Artefakten fassen über „**manifest**"-Abhängigkeiten Pakete, Klassen, ... zu verteilbaren Dateien (z.B. als jar-Dateien oder Binärdateien) zusammen.



Verteilungsdiagramm (deployment diagram) als Netzwerkbeschreibung:

Visual Paradigm for UML Community Edition [not for commercial use]

FrontOfficePC

BackOfficePC

Verbindung (Association) zwischen Knoten

Ethernet (LAN)

Firewall

DSL

Server

Knoten (Node)

Es gibt zwei PCs und einen (Datenbank-)Server, die über Ethernet (LAN) miteinander über eine Firewall mit dem Internetverbunden sind und über Fire. Ein Bus (wie LAN) kann nur als Knoten „modelliert“ werden, da Verbindungen immer bidirektional sind.



Verteilungsdiagramme für verschiedene Software-Verteilungsszenarien:



Verschiedene Verteilungsmöglichkeiten:

- **Ultra-Thin Client**: die gesamte Software läuft auf dem Server, Anzeige auf dem Client-PC = Terminal erfolgt beispielsweise über X-Protokoll (überträgt Fensterinhalte vom Server zum Client und Benutzereingaben vom Client zum Server)
- **Thin Client**: die Benutzeroberflächensoftware ist vollständig auf dem Client, die Kommunikation mit der Anwendungsschicht (und der darunterliegenden Datenbank) erfolgt über Corba oder Java "remote method invocation" oder …
- **Fat Client**: Benutzeroberfläche und Anwendungsschicht befinden sich auf dem Client, Datenbank-API (application interface) regelt Kommunikation mit Datenbank auf dem Server
- **Ultra-Fat Client**: die gesamte Anwendung einschließlich der Datenbank (oder Teilen der Datenbank) befinden sich auf dem Client, ein sogenanntes verteiltes Datenbanksystem regelt Zugriff auf Datenbanken auf verschiedenen Rechnern
- ...



8.4 Softwareevolution und Refactoring

„Refactoring" ist Evolution bzw. **Sanierung von Software in kleinen Schritten**, ohne das Verhalten der betroffenen Software zu ändern. Regressionstests (nach jeder Änderung neu durchgeführte Tests) werden eingesetzt, um nach jedem kleinen Umbauschritt sicherzustellen, dass sich Softwareverhalten nicht geändert hat.

Ziele des Refactorings:

- ⇨ Verständlichkeit von Software erhöhen
- ⇨ Änderungen und Erweiterungen von Software erleichtern
- ⇨ [effizienzsteigernde Maßnahmen erleichtern]
- ⇨ Ähnliche (identische) Codestellen zusammenfassen
- ⇨ durch Metriken/Analysen als fragwürdig erkannten Code „ausmerzen"
- ⇨ Debugging von Software vereinfachen
- ⇨ dem Alterungsprozess von Software entgegenwirken



Gründe, aus denen Refactoring nicht eingesetzt wird:

- wegen Termindruck ist angeblich keine Zeit „nutzlose“ Aufräumarbeiten durchzuführen
  - ⇨ aber: anschließende Erweiterungen, Fehlersuche ... gehen schneller
- Refactoring führt zu (ggf. tatsächlich erst einmal) ineffizienteren Programmen
  - ⇨ aber: effizienzsteigernde Programmtransformationen sind anschließend einfacher durchzuführen
- „never touch a running system“: Umbauten könnten unbeabsichtigt Programmverhalten ändern und damit neue Fehler einbauen:
  - ⇨ aber: transformierte Programme sind leichter zu verstehen und damit leichter änderbar, ...
  - ⇨ aber: Regressionstests helfen dabei, unbeabsichtigte Verhaltensänderungen zu erkennen
  - ⇨ aber: Refactoring-Werkzeuge helfen bei der Durchführung verhaltensbewahrender Programmtransformationen



Beispielprogramm - Kernteile eines Videoverleihsystems:

Achtung: Refactoring eines so kleinen Programmes ist weitgehend sinnlos; gezeigt werden hier nur die Prinzipien des Refactorings an einem überschaubaren Java-Beispiel aus [Fo00] (es handelt sich um ein (Video-)Verleihverwaltungssystem, also ein ähnliches Beispiel wie wie „unser“ MVRS). Bei großen Softwaresystemen ist die Vorgehensweise aber genau die gleiche.

```
public class Rental {

    private Movie _movie;
    private int _daysRented;

    public Rental (Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    };

    public int getDaysRented() {
        return _daysRented;
    };

    public Movie getMovie() {
        return _movie;
    };

}
```



Beispielprogramm - Klasse Movie:

```
public class Movie {

    public static final int CHILDREN = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;
        // regular movie or for children or ...

    public Movie (String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    };

    public int getPriceCode() {
        return _priceCode;
    };

    public void setPriceCode(int arg) {
        _priceCode = arg;
    };

    public String getTitle () {
        return _title;
    };

}
```



Beispielprogramm - Klasse Customer:

```
public class Customer {
    private String _name;
    private Vector _rentals = new Vector();

    public Customer (String name) {
        _name = name;
    };

    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    };

    public String getName() {
        return _name;
    };

    public String statement() {
        // creates listing (text output) with all available information about given customer

        ...

    };

    public String htmlStatement() {
        // creates html output with all available information about given customer

        ...

    };
```



Beispielprogramm - Methode statement:

```
public String statement() {
    // creates listing (text output) with all available information about given customer

    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        // determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDREN:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                break;
        };
```



Beispielprogramm - Fortsetzung:

```
        // still inside the while statement

            //  add frequent renter points
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1)
                frequentRenterPoints++;

            // show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;

        };

        // add footer lines
        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
        return result;

    };

    public String htmlStatement() {
        // creates html output with all available information about given customer
        // diese Methode sieht fast wie statement-Methode aus

        ...

    };

};
```



Kritik an dem erstellten Programm:

- Customer-Klasse führt dauernd Berechnungen über Daten aus Movie durch

Zusammenfassung der Kritik:

- „no separation of concern / low cohesion“ bei Customer-Klasse
- „high coupling“ zwischen Customer und Movie



Erster Refactoring-Schritt - „ExtractMethod“:

```
...
while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    // determine amounts for each line
    switch (each.getMovie().getPriceCode()) {
        ...
    };

    //  add frequent renter points
    ...
};
```

Methode statement nach Refactoring:

```
...
while (rentals.hasMoreElements()) {
    double thisAmount = 0;
    Rental each = (Rental) rentals.nextElement();

    thisAmount = amountFor(thisAmount, each);

    //  add frequent renter points
    ...
};
```

(In `amountFor(thisAmount, each)` ist „thisAmount, “ durchgestrichen.)



Extrahierte Methode amountFor:

```
private double amountFor(double thisAmount, Rental each) {
// determine amounts for each line
    double thisAmount = 0;

    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDREN:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
    };

    return thisAmount;
};
```

Anmerkung: die Methode ist mit Eingabeparameter thisAmount von CASE-Tool automatisch extrahiert worden; da aber Parameter thisAmount immer mit Wert 0 vor Aufruf initialisiert wird, kann man eine lokale Variable daraus machen.



Zweiter Refactoring-Schritt - „Rename...“:

„Vernünftige“ Namen für Variablen, Methoden, Klasssen, ... sind wichtig für die Lesbarkeit von Programmen. Deshalb sind Umbenennungen wie each in aRental oder thisAmount in result kein überflüssiger Luxus:

```
private double amountFor(Rental aRental) {
// determine amounts for each rental object

    double result = 0;

    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented() - 2) * 1.5;
            break;
        case Movie.CHILDREN:
            result += 1.5;
            if (aRental.getDaysRented() > 3)
                result += (aRental.getDaysRented() - 3) * 1.5;
            break;
    };

    return result;
};
```



Dritter Refactoring-Schritt - „MoveMethod“:

Die Methode amountFor der Klasse Customer führt ausschließlich Berechnungen auf Attributen der Klasse Rental durch. Sie ist deshalb in der falschen Klasse deklariert. Eine Verschiebung der Methode erfolgt in vier Schritten (mit Test dazwischen):

1. die Klasse Rental erhält eine Kopie der Methode amountFor der Klasse Customer
2. Implementierung alter Methode wird durch Aufruf der neuen Methode ersetzt
3. ggf. werden Aufrufe der alten Methode durch Aufrufe der neuen Methode ersetzt
4. alte Methode wird gelöscht, sobald sie nicht mehr aufgerufen wird

```
public class Customer { ...

    private double amountFor(Rental aRental) {
        return aRental.getCharge();
    };
};

public class Rental { ...

    public double getCharge() {
        double result = 0;
        switch (getMovie().getPriceCode()) {
            ...
        };
        return result;
    };
```

(Im Original sind die Zeilen „private double amountFor(Rental aRental) {“, „return aRental.getCharge();“ und „};“ durchgestrichen.)



Die veränderte Methode statement von Customer:

```
public String statement() {
    // creates listing (text output) with all available information about given customer

    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        thisAmount = each.getCharge();
        frequentRenterPoints += each.getFrequentRenterPoints();
        // Extract- und MoveMethod auch für frequentRenterPoints-Berechnung durchgeführt

        // show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;

    };

    // add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
    return result;

};
```



Beispielprogramm - die veränderte Klasse Rental komplett:

```
public class Rental {
    private Movie _movie; private int _daysRented;
    public Rental (Movie movie, int daysRented) {
        _movie = movie; daysRented = daysRented;
    };
    public int getDaysRented() {
        return _daysRented;
    };
    public Movie getMovie() {
        return _movie;
    };
    public double getCharge() {
        double result = 0;
        switch (getMovie().getPriceCode()) { ... };
        return result;
    };
    public int getFrequentRenterPoints() {
        if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) && (getDaysRented() > 1))
            return 2;
        else
            return 1;
    };
};
```



Verbleibende Kritik am Beispielprogramm:

- die Schleife der Methode statement vermischt immer noch Erstellung eines Ausdrucks und Berechnung von Gebühren und Bonuspunkten
- Ausleihgebühren eines Kunden und Bonuspunkte lassen sich nicht direkt abfragen (nur aus Ergebnis von statement extrahieren)
- die Methode getCharge enthält immer noch ein switch-Statement
- ...

Notwendige Umbauten:

1. Berechnung von totalAmount und frequentRenterPoints in eigene Methoden extrahieren
2. resultierende Ineffizienzen durch weitere Umbauten beheben
3. später dann die Fallunterscheidungen in den Methoden getCharge und frequent-RenterPoints durch Polymorphie ersetzen



Die Methode statement mit extrahierten Berechnungen:

```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental aRental = (Rental) rentals.nextElement();
        result += "\t" + aRental.getMovie().getTitle() + "\t" + String.valueOf(aRental.getCharge()) + "\n";
    };
    result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " + String.valueOf(getTotalFrequentRenterPoints()) + " frequent renter points";
    return result;
};

public int getTotalCharge() {
    int result = 0; Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        result += ((Rental) rentals.nextElement()).getCharge();
    };
    return result;
};

public int getTotalFrequentRenterPoints() {
    int result = 0; Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        result += ((Rental) rentals.nextElement()).getFrequentRenterPoints();
    };
    return result;
};
```



Nachteile der neuen Implementierung;

- die Methode getCharge wird für jedes Rental-bzw. Movie-Objekt zweimal aufgerufen (einmal in statement und einmal in getTotalCharge)
- anstelle einer Schleife über alle Rental-Objekte gibt es nunmehr in drei verschiedenen Methoden jeweils eine eigene Schleife

Refactoring-Schritte zur Effizienzsteigerung:

- Charge eines Rental-Objektes wird bei Erzeugung berechnet und gespeichert (getCharge liefert dann nur gespeicherten Wert zurück); dieser Schritt nennt sich „Caching“ von Methoden- bzw. Funktionswerten in Attributen
- gleiches könnte man für TotalCharge und TotalFrequentRenterPoints tun: für neuen Kunden sind beide Werte gleich 0, sie werden für jedes neu hinzukommende Rental-Objekt danach erhöht
- Achtung: effizienzsteigernde Refactoring-Schritte **nur** dann durchführen, wenn Laufzeitmessungen auf Probleme hinweisen



Vorbereitung für Elimination von „conditional code“:

Verschieben der Methoden getCharge und getFrequentRenterPoints von Klasse Rental in Klasse Movie, da sie vom PriceCode von Movie abhängen:

```
public class Rental {
    private Movie _movie; private int _daysRented;
    public Rental (Movie movie, int daysRented) {
        _movie = movie; daysRented = daysRented;
    };
    public int getDaysRented() {
        return _daysRented;
    };
    public Movie getMovie() {
        return _movie;
    };
    public double getCharge() {
        return _movie.getCharge(_daysRented);
    };
    public int getFrequentRenterPoints() {
        return _movie.getFrequentRenterPoints(_daysRented);
    };
};
```



Die veränderte Klasse Movie mit getCharge und getFrequentRenterPoints:

```
public class Movie {
    ...
    public double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDREN:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        };
        return result;
    };

    public int getFrequentRenterPoints(int daysRented) {
        return ... ;
    };

};
```



Effizientere Klasse Rental mit „Caching“ von Methodenwerten:

```
public class Rental {
    private Movie _movie;
    private int _daysRented;
    private double _charge;
    private int _frequentRenterPoints;

    public Rental (Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
        _charge = movie.getCharge(daysRented);
        _frequentRenterPoints = movie.getFrequentRenterPoints(daysRented);
    };

    public double getCharge() {
        return _charge;
    };

    public int getFrequentRenterPoints() {
        return _frequentRenterPoints;
    };

    ...

};
```



Noch notwendiger Umbau - Elimination des Switch-Statements:

```
public class Movie {
    ... ;
    public double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDREN:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        };
        return result;
    };

};
```

Achtung: Ansatz mit neuen Klassen RegularMovie, NewMovie und ChildrenMovie scheitert, da sich PriceCode während Lebenszeit von Movie-Objekt ändert!



Refactoring-Schritt „Replace Type Code by State Class“ für Movie:

```
public class Movie {
    public static final int CHILDREN = 2; ...
    private String _title;  private MovieState _movieState;

    public Movie (String title, int priceCode) {
        _title = title; setPriceCode(priceCode);
    };

    public void setPriceCode(int arg) {
        switch (arg) {
            case Movie.REGULAR:
                _movieState = new RegularMovie();
                break;
            case Movie.NEW_RELEASE:
                _movieState = new NewMovie();
                break;
            case Movie.CHILDREN:
                _movieState = new ChildrenMovie();
                break;
    };

    public int getPriceCode() {
        return _movieState.getPriceCode();
    };

    public double getCharge(int daysRented) {  ... };
        // hat sich überhaupt nicht geändert, da alle Zugriffe auf PriceCode über Methode getPriceCode realisiert waren
```



Die neuen State-Klassen für Movie:

```
abstract class MovieState {
    abstract int getPriceCode();
};

class RegularMovie extends MovieState {
    int getPriceCode() {
        return Movie.REGULAR;
    };
};

class NewMovie extends MovieState {
    int getPriceCode() {
        return Movie.NEW_RELEASE;
    };
};

class ChildrenMovie extends MovieState {
    int getPriceCode() {
        return Movie.CHILDREN;
    };
};
```

Achtung: die neuen Klassen sind noch ziemlich sinnlos, aber man hat einen „vernünftigen“ Zwischenzustand für Programmtests erreicht. Denn: niemals zu große Refactoring-Schritte auf einmal durchführen, sondern viele kleine (durch Werkzeug durchgeführte) Schritte mit eingestreuten Tests!!!



Refactoring-Schritt „MoveMethod“ von Movie nach MovieState:

```
public class Movie { ...
    public double getCharge(int daysRented) {
        return _movieState.getCharge(daysRented);
    };
};

abstract class MovieState { ...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDREN:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        };
        return result;
    };
};
```



Refactoring-Schritt „ReplaceConditional with Polymorphism“, ... :

```
abstract class MovieState { ...
    abstract double getCharge(int daysRented);
};

class RegularMovie extends MovieState { ...
    double getCharge(int daysRented);
        double result = 2;
        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;
        return result;
    };
};

class NewMovie extends MovieState { ...
    double getCharge(int daysRented);
        return daysRented * 3;
    };
};

class ChildrenMovie extends MovieState { ...
    double getCharge(int daysRented);
        double result = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        return result;
    };
};
```



„ReplaceConditional with Polymorphism“ - Fortsetzung :

```
abstract class MovieState { ...
    double getFrequentRenterPoints(int daysRented) {
        return 1;
    };
};

// the points policy introduced above is currently valid for all movies except of new releases

class NewMovie extends MovieState { ...
    double getFrequentRenterPoints(int daysRented) {
        return (daysRented > 1) ? 2 : 1;
    };
};

public class Movie { ...
    public double getFrequentRenterPoints(int daysRented) {
        return _movieState.getFrequentRenterPoints(daysRented);
    };
};
```



Gründe für Refactoring - 1:

- **duplicate code**: fast derselbe Code findet sich an n Stellen (Fehler an n Stellen beheben, ...) - wie etwa bei statement und htmlStatement
  - ⇨ „Extract Method“ um identische Codeteile an einer Stelle zu konzentrieren
- **long class/method**: eine Klasse oder Methode ist sehr lang oder enthält Schleifen mit umfangreichem Rumpf oder ... - wie bei statement
  - ⇨ „Extract Class“, „Extract Method“ um Code zu zerschlagen
- **long parameter list**: eine Methode besitzt sehr viele Parameter (kann z.B. für Ergebnis von ExtractMethod gelten)
  - ⇨ „Replace Parameter with Method“ ersetzt Par. durch Methodenaufruf
  - ⇨ „Introduce Parameter Object“ fasst mehrere Parameter zusammen
- **divergent change (low cohesion)**: logisch verschiedene Programmänderungen werden immer wieder in der selben Klasse durchgeführt - wie bei statement
  - ⇨ „Extract Class“ zerlegt Klasse in verschiedene Klassen für jeweils einen logischen Zweck



Gründe für Refactoring - 2:

- **shotgun surgery (high coupling)**: das Gegenteil von „divergent change“ - eine logische Programmänderung erfordert Umbauten an vielen Klassen, Methoden, ...
  - ⇨ „Move Method“ für Konzentration von Änderungsstellen
  - ⇨ „Move Attribute“ für Konzentration von Änderungsstellen
- **feature envy**: eine Klasse ist mehr an den Attributen/Feldern einer anderen Klassen interessiert als diese selbst - z.B. statement an Attributen von Rental
  - ⇨ „Move Method“ bringt „neidische“ Methoden zu benutzten Attributen
- **switch statements**: wenn komplexe wiederkehrende bedingte Anweisungen anstelle von Unterklassen und Polymorphie eingesetzt werden - z.B. in Movie
  - ⇨ „Replace Type Code with State“
  - ⇨ „Replace Conditional with Polymorphism“
- **lazy class**: eine Klasse hat nicht mehr genug Aufgaben (ggf. wegen Refactoring)
  - ⇨ „Collapse Hierarchy“ um nutzlose Klasse in Oberklasse aufgehen zu lassen



Gründe für Refactoring - 3:

- **message chains**: eine Methode holt sich mit „gettern“ ein Objekt eines Objekts eines ... , um darauf eine Berechnung durchzuführen
  - ⇨ „Extract Method“ und „Move Method“ um Berechnung zum geholten Objekt zu bringen (anstelle dieses Objekt zur Berechnung)
- **middle man**: fast alle Methoden einer Klasse delegieren ihre Aufrufe an Methoden einer anderen Klasse - kann z.B. Folge der Elimination von message chains sein
  - ⇨ „Inline Method“ um Methodenaufruf durch Implementierung zu ersetzen
  - ⇨ ...
- **refused bequest**: Unterklasse C1 benötigt viele Methoden und Attribute ihrer Oberklasse C nicht
  - ⇨ neue Oberklasse C’ der Klasse C einführen und mit „Pull Up ... “ nicht in C1 benötigte Methoden und Attribute von C nach C’ bringen sowie C1 in Unterklasse von C’ statt C umwandeln



Gründe für Refactoring - 4:

- **comments**: Kommentare im Methodenrumpf sind notwendig um diesen zu verstehen
  - ⇨ „Extract Method“ führt eigene Methode mit vernünftigem Namen und Kommentierung für schwer verständlichen Methodenteil ein
- **data class**: Klasse, die nur Daten enthält und auf diesen keine Berechnungen durchführt - insbesondere dann, wenn Attribute/Felder public sind
  - ⇨ „Encapsulate Attribute“ führt Get- und Set-Methoden für nunmehr private Attribute ein
  - ⇨ „Move Method“ bringt Berechnungen zu den Attributen

  Achtung: widerspricht etwas der Empfehlung zur strikten Trennung von Daten-haltungs- und Berechnungsklassen (siehe <<Entity>>-Steoreotyp); solche Klassen machen Sinn, falls sie von einem „Speichermechanismus“ abstrahieren
- ...



Aufbau der Beschreibung von Refactoring-Schritten:

- **Name** des Refactoring-Schrittes
- **Vorbedingung**, die für die Anwendung erfüllt sein muss
- **Kurzbeschreibung** des Refactoring-Schrittes
- **Minimalbeispiel**, das das Prinzip klar macht
- **Motivation** für die Durchführung einer solchen Programmtransformation
- **Kochrezept** für die Durchführung der Transformation (mechanics) mit Einzelschritten, Fallstricken, zu überprüfenden Randbedingungen, ...
- weitere **Beispiel(e)** für die Anwendung der Transformation



Extract Method:

- **Vorbedingung**: es gibt ein Codefragment, das sich vom umstehenden Code abhebt
- **Kurzbeschreibung**: Codefragment wird zu Methode mit sinnvollem Namen
- **Beispiel - vorher**:

```
void printOwing(double amount) {
    printBanner();
    // print all the details
    System.out.println ("name:" + _name);
    System.out.println ("amount:" + amount);
};
```

Beispiel - nachher:

```
void printOwing(double amount) {
    printBanner(); printDetails(amount);
};

void printDetails(double amount) {
    // print all the details
    System.out.println ("name:" + _name);
    System.out.println ("amount:" + amount);
};
```



Extract Method - Kochrezept:

1. erzeuge eine neue Methode mit sinnvollem Namen in der selben Klasse
2. kopiere den zu extrahierenden Code von der alten zur neuen Methode
3. suche im extrahierten Code nach Parametern und lokalen Variablen, die nicht mit dem extrahierten Code kopiert wurden
4. führe für alle diese Parameter und Variablen in der neuen Methode Parameter ein
5. stelle fest, ob die betroffenen Parameter und Variablen in der neuen Methode verändert werden
6. wird kein Parameter und keine Variable verändert, ist die neue Methode void
7. wird genau ein Parameter oder eine Variable verändert, so wird sie mit return an die Aufrufstelle zurückgegeben, sonst ...
8. übersetze geändertes Programm und behebe Fehler
9. ersetze kopierten Code durch Aufruf der neuen Methode
10. lösche überflüssige Variablen, deren Anwendungsstellen alle ausgelagert wurden
11. übersetze und teste geändertes Programm



Große Refactorings bzw. Softwaresanierung:

Bislang wurden nur Refactorings vorgestellt, die kleine und im wesentliche lokale Änderungen an einem Programm vornehmen. Oft muss aber die gesamte Programm-struktur im großen Umfang „saniert“ werden. Dann spricht von „großen Refactorings“, siehe [RL04]. Diese Sanierungsmaßnahmen müssen

- sorgfältig geplant werden (siehe Projektmanagement in „Software Engineering I“) und in einen iterativen Softwareentwicklungsprozess integriert werden (als eigene Refactoring-Iterationen oder als Teil von normalen Iterationen)
- es muss dafür Kostenschätzung durchgeführt werden (normale Kostenschätzung für Neuentwicklungen ist leider nicht anwendbar)
- ggf. als eigener „Branch“ im Versionsverwaltungssystem geführt werden, damit funktionale Weiterentwicklung des Systems und Refactoring sich nicht stören (Problem: wann werden Änderungen integriert?!)



Bewahrung von Teilsystemschnittstellen beim Refactoring:

Oft hat der Entwickler eines Teilsystems (einer Bibliothek, eines Rahmenwerkes, ...) nicht Zugriff auf den Code der Anwendungen, die sein Teilsystem verwenden. Damit kann er im Zuge des Refactorings seines Teilsystems eigentlich keine Änderungen an der Schnittstelle seines Teilsystems durchführen ohne alle Anwender zu Umbauten zu zwingen (Details hierzu in [RL04])

- ❑ deshalb versucht man neue Versionen eines Teilsystems (beim Refactoring, aber nicht nur dabei) soweit möglich „abwärtskompatibel“ zu gestalten
- ❑ baut man „Umleitungen“ ein, die alte nicht mehr erwünschte Schnittstellenanteile auf neuere Realisierung abbilden und damit Anwender nicht zum sofortigen Umbau ihrer Anwendungen zwingen
- ❑ alte Klassen und Methoden werden als nicht mehr zu benutzend markiert (in Java „deprecated“); diese Markierung führen zu Compilerwarnungen

☺ **Vorteil**: Anwender hat Zeit, um notwendige Umbauten durchzuführen

☹ **Nachteil**: neue Softwarearchitektur ist teilweise schlechter als alte, weil Altlasten nun zusätzlich zur sanierten/neuen Lösung weiter existieren



Beispiele für nicht abwärtskompatible Schnittstellenänderungen:

- Änderungen an Klassen:
  - Umbenennung oder Verschieben in ein anderes Paket: Klasse kann nicht mehr benutzt werden
  - Klasse löschen: kann nicht mehr benutzt werden
  - Klasse hinzufügen: ggf. Namenskonflikte mit gleichnamigen Klassen
  - Oberklassen hinzufügen: neue abstrakte Methoden zu realisieren
  - Oberklassen löschen: bislang geerbte Methoden und Attribute fehlen
  - ...
- Änderungen an Interfaces/Methoden von Klassen:
  - Umbenennen führt zu ähnlichen Problemen wie Umbenennen von Klassen
  - Löschen: betroffene Methoden können nicht mehr gerufen werden
  - Hinzufügen eines Interfaces: betroffene Methoden müssen erst noch neu implementiert werden; es kann Namenskonflikte geben



Beispiele für nicht abwärtskompatible Schnittstellenänderungen - 2:

- Änderungen an Methoden:
  - Hinzufügen abstrakter Methode: muss in Unterklassen realisiert werden
  - Umbenennen, Löschen, ...
  - Sichtbarkeit einschränken: Methode kann ggf. nicht mehr gerufen werden
  - Sichtbarkeit erweitern - von private auf protected: in Unterklassen kann es zu Konflikten mit gleichnamigen Methoden kommen
  - Methode von „nicht final“ auf „final“ setzen
  - ...
- Änderungen an Ausnahmen von Methoden:
  - Umbenennen: ...
  - Löschen: Aufrufer der Methode dürfen die Ausnahme nicht mehr abfangen, Redefinitionen der Methode können Ausnahme nicht mehr erwecken
  - Hinzufügen: Aufrufe der Methode müssen Ausnahme abfangen
- viele weitere Beispiele für inkompatible Schnittstellenänderungen in [RL04]



8.5 Design-Pattern und Anti-Pattern

Bisher haben wir durch Refactoring systematisch „schlechte“ Programm in „gute“ umgebaut. Dabei wurde die Frage, wie man „schlecht“ und „gut“ definiert nur angerissen und der Schwerpunkt auf die Beschreibung von Umbaumaßnahmen gelegt. Hier führen wir

- **Design-Pattern** (**gute Entwurfsmuster**) als systematische Beschreibungsmittel für „gute“ Programmkonstruktionsprinzipien
- **Anti-Pattern** (**schlechte Entwurfsmuster**) als systematische Beschreibungsmittel für „schlechte“ Programmkonstruktionsprinzipien

ein. Zu beachten gilt, dass Design-Pattern ursprünglich von dem Architekten Christopher Alexander eingeführt wurden [Al77] und sich nicht nur für den Entwurf guter Programme eignen, sondern auch für die Analysephase oder das Projektmanagement. Man spricht dann von Analyse-Pattern, ... Zudem werden Entwurfsmuster in der Architektur, dem Maschinenbau, ... in Form von Entwurfsregeln eingesetzt. Das gilt ebenfalls für das Gegenstück der Design-Pattern, die Anti-Pattern.



Unser Videothek-Beispiel - aktueller Stand nach Refactoring:

Sieht man von der noch nicht vollzogenen Einführung von „State"-Klassen anstelle des Preiscodes ein, haben alle bisherigen Refactoring-Schritte die Klassendiagramm-Struktur des Programms nicht verändert. Bei Hinzufügen neuer Funktionalität werden wir (deshalb) erneut in Probleme laufen und nunmehr den Einsatz bekannter Entwurfs-muster demonstrieren.



Gewünschte Programmerweiterungen:

- Es soll eine Klasse Videoshop eingeführt werden, die alle Kunden plus allgemeine Daten und Verantwortlichkeiten aufnimmt.
- Zudem brauchen wir eine Gruppierung von Ausleihvorgängen nach Jahren, ...

Eingesetzte Design-Pattern:

- Für die neue Klasse Videoshop mit genau einer Instanz das „**Singleton**"-Pattern.
- Für die Realisierung einer Hierarchie auf Rental-Objekten werden wir das „**Composite**"-Pattern verwenden.
- Für die Realisierung des PriceCodes das bereits im vorigen Abschnitt vorgestellte „**State**"-Pattern, das neue „**Flyweight**"- und das „**Factory**"-Pattern.
- Für die Traversierung der Rental-Hierarchie das bereits bekannte „**Iterator**"-Pattern (siehe etwa Java).
- benfalls bereits verwendet wurde das „**Chain of Responsibility**"-Pattern.
- Für die Trennung von „Model" und „View" das „**Observer**"-Pattern.



Überblick über klassische Design-Pattern:

Das klassische Buch zum Thema Software-Design-Pattern wurde von der „**Gang of Four**“ (Gamma, Helm, Johnson, Vlissides) geschrieben. Sie führen in [GH+94] etwa 22 Design-Pattern (in drei Gruppen) ein. Einen Ausschnitt aus der damit eingeführten Pattern-Sprache bzw. Familie zeigt folgende Grafik:

Iterator
Structural Patterns
für Durchlaufen der Kinder
„sharing“ von
Teilhierarchien
Flyweight
Composite
„sharing“ von
Zuständen
definiert Chain of Responsibility
erzeugt
Flyweights
...
Chain of Responsibility
State
Observer
Behavioral Pattern
Factory
Singleton
verwaltet Singletons
Creational Patterns



Geändertes Videoverleih-Programm (Ausschnitt):



Aufbau eines Entwurfsmusters in Anlehnung an [GH+94]:

- **Name + Synonyme**: Namen unter denen das Muster bekannt ist
- Klassifikation: Einordnung in vorgestelltes Schema (Pattern-Sprache)
- **Absicht**: Beschreibung des Hauptzwecks in einem Satz
- **Motivation**: wofür ist Muster geeignet bzw. was für Probleme soll es lösen
- **Anwendbarkeit**: in welchen Situationen ist das Muster (besonders) gut einsetzbar
- **Struktur**: generisches Klassendiagramm für Entwurfsmuster
- **Kollaboration**: Zusammenspiel der Klassen als Interaktionsdiagramm oder Text
- **Konsequenzen**: wie erreicht das Muster die (in Motivation) aufgeführten Ziele
- **Implementierung**: Implementierungshinweise (mit weiteren Codebeispielen); Hinweise auf Fallstricke, programmiersprachenspezifische Aspekte etc.
- **Einfaches Beispiel**: ein Beispiel, das die Anwendung des Musters verdeutlicht
- **Bekannte Anwendungen**: Software in der Muster erfolgreich eingesetzt wurde
- **Verwandte Muster**: Verweise auf Muster, die ähnliche Probleme addressieren



Das Entwurfsmuster „Iterator“:

- **Name**: Iterator
- **Synonym**: Cursor im Datenbankbereich oder auch Enumeration
- **Absicht**: erlaubt geschachtelte und/oder beliebig viele Iterationen über Menge/Liste von Objekten (ggf. auch über Objekthierarchien)
- **Motivation**: oft wird das Traversieren der Elemente einer Datenstruktur aus softwaretechnischer Sicht falsch realisiert:
  - ⇨ Anwendung erhält Zeiger auf Element in Datenstruktur: damit wird Datenstruktur offengelegt und Änderungen an Datenstruktur schaffen Probleme
  - ⇨ Datenstruktur hat Operationen first und next und merkt sich intern einen Zeiger auf gerade aktuelles Element: damit sind mehrere Iterationen über derselben Datenstruktur gleichzeitig nicht möglich
- **Anwendbarkeit**: ...



Fortsetzung des „Iterator“-Entwurfsmusters:

- **Struktur** (Ellipse = instantiierbares Teildiagramm):

Element

+Element

0..*

0..*

Index i ist Index in Vector in Aggregate oder Zeiger auf mit nächstem Element verkettetes Element.

Aggregate

+Aggregate
+iterate:Iterator
+addChild:void
first:Index
next:Index
current:Element

1

0..*

Iterator

-_current:Index

+Iterator
+first:void
+next:void
+current:Element
+isDone:boolean

first, next und current haben in Java package local Sichtbarkeit; in C++ würde man private Sichtbarkeit und friend-Beziehung von Iteratur zu Aggregate einsetzen. current und next haben (nicht sichtbaren) Parameter Index i.

Iterator



Fortsetzung des „Iterator“-Entwurfsmusters:

Klassen in der Struktur (Teilnehmer/Rollen des Musters):

- ⇨ Aggregate: eine Kollektion bzw. Ansammlung von Elementen, auf der mehrere parallele Durchläufe bzw. Iterationen durchgeführt werden sollen; diese Klasse erzeugt neue Iterator-Objekte für diesen Zweck
- ⇨ Element: die Elemente von Aggregate, die bei einem Durchlauf zurück-gegeben werden
- ⇨ Iterator: Objekte dieser Klassen „merken“ sich den Stand der Iteration, sodass mit den zur Verfügung stehenden Methoden von Aggregate eine unterbrochene Iteration jederzeit wieder fortgesetzt werden kan



Fortsetzung des „Iterator“-Entwurfsmusters:

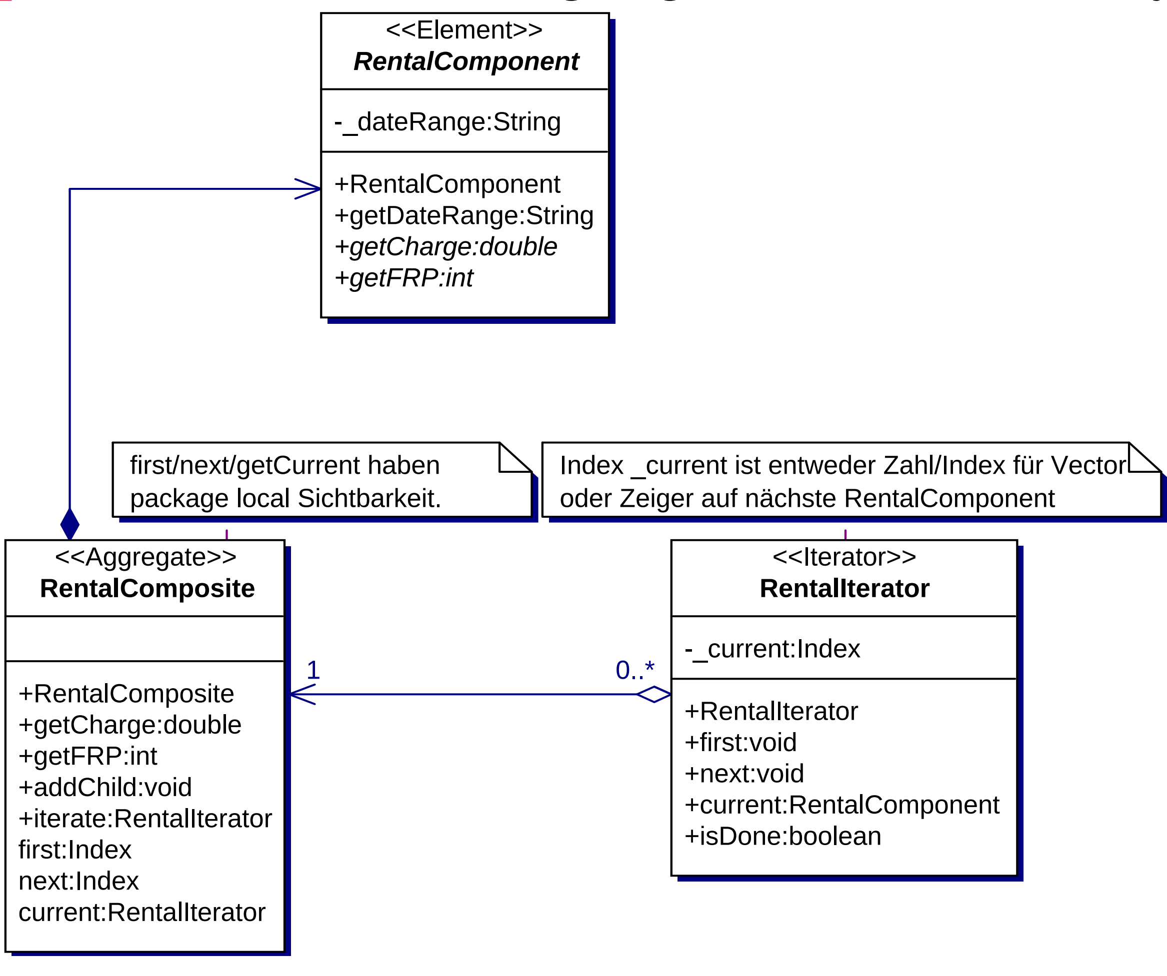
- **Kollaboration** (hier textuell definiert und nicht als Sequenzdiagramm):
  1. Mit der Methode iterate wird neuer Iterator zu einem Aggregate-Objekt erzeugt. Sie liefert ein Iterator-Objekt als Ergebnis, das auf Aggregate zeigt.
  2. Der Durchlauf durch alle Elemente des Aggregats wird mit dem Aufruf von first auf dem Iterator-Objekt gestartet; es wird dabei _current = first auf Aggregate aufgerufen und dabei Attribut _current für erstes Element gesetzt.
  3. Mit current auf Iterator-Objekt fordert man das aktuelle Element an; hierfür wird current(_current) auf Aggregate aufgerufen,was das aktuelle Element-Objekt zurückliefert (gibt es kein weiteres Objekt mehr, so wird das null-Objekt zurückgegeben).
  4. Mit next-Aufruf auf Iterator-Objekt wird _current = next(_current)-Aufruf auf Aggregate ausgelöst und somit der Index auf nächstes Element des gesetzt.

  Anmerkung: wird Aggregat als Array/Vector realisiert, so ist Index eine Zahl, die durch next jeweils um eins hochgezählt wird. Wird Aggregat als verzeigerte Datenstruktur (Liste, ...) realisiert, so zeigt Index immer auf das nächste Element.



Fortsetzung des „Iterator“-Entwurfsmusters:

- **Beispiel** für Iterator-Instanziierung - eigene Notation mit Stereotypen:





❑ Beispiel für Iterator-Instanziierung - **UML-Notation**:

RentalComponent
-_dateRange:String
+RentalComponent
+getDateRange:String
+getCharge:double
+getFRP:int

Element

Aggregate

Iterator

RentalComposite
+RentalComposite
+getCharge:double
+getFRP:int
+addChild:void
+iterate:RentalIterator
first:Index
next:Index
current:RentalIterator

1

0..*

RentalIterator
-_current:Index
+RentalIterator
+first:void
+next:void
+current:RentalComponent
+isDone:boolean



Fortsetzung des „Iterator“-Entwurfsmusters:

- **Konsequenzen** (Vorteile):
  - die Verwendung der Iterator-Schnittstelle sollte einfacher als die Verwendung der Schnittstelle von Aggregate sein
  - einheitlich gestalte Schnittstelle für Iterationen über beliebigen Aggregaten
  - in der Implementierung können verschiedenen Durchlaufstrategien versteckt werden (Filtern, rückwärts durchlaufen, ...)
  - mehrere Iterationen können gleichzeitig aktiv sein
  - Varianten des Musters unterstützen sogar die Verschränkung von Hinzufügen/Löschen von Elementen auf einem Aggregat, auf dem eine Iteration gerade läuft



Fortsetzung des „Iterator“-Entwurfsmusters:

- **Implementierungshinweise:**
  - während der Iteration sollte das Aggregat nicht verändert werden (insbesondere das current-Objekt nicht aus dem Aggregat gelöscht werden)
  - will man Iterationen auf sich ändernden Aggregaten unterstützen, so muss das Aggregat die Möglichkeit besitzen, den Iterator von Löschungen zu informieren (oder Löschen nicht wirklich durchführen, sondern nur Markierungen setzen)
  - im Beispiel wurde der Iterator auf einer Hierarchie von Rental-Objekten definiert; trotzdem liefert im Beispiel der Iterator nur die direkten Kinder eines CompositeRental-Objektes
  - es kann auch leicht ein Iterator definiert werden, der die gesamte (Teil-)Hierarchie unterhalb eines CompositeRental-Objektes durchläuft
- **Verwandte Muster**: Composite, ...



Das Entwurfsmuster „Composite“ (stark verkürzte Beschreibung):

- **Name**: Composite
- **Absicht**: die Anordnung von Objekten in Hierarchien ist ein Standardproblem beim Entwurf von Softwaresystemen; die vorgeschlagene Lösung behandelt zusammengesetzte Objekte und atomare Objekte einer Hierarchie gleich.
- **Motivation**: oft können Objekthierarchien beliebig tief geschachtelt werden und auf den zusammengesetzten wie auf den atomaren Objekten (Blättern der Hierarchie) werden dieselben Operationen zur Verfügung gestellt. Beispiele für solche Operationen sind:
  - Löschen von Unterbäumen und Blättern einer Hierarchie
  - Ausgeben (Drucken, Malen, ...) einer Hierarchie
  - Aufsummieren von Werten auf einer Hierarchie (bei uns Charge und F(requent)R(enter)P(oints)
  - ...



Struktur des „Composite“-Musters:



Beispiel für „Composite“-Muster:



Das Entwurfsmuster „State“ (verkürzte Beschreibung):

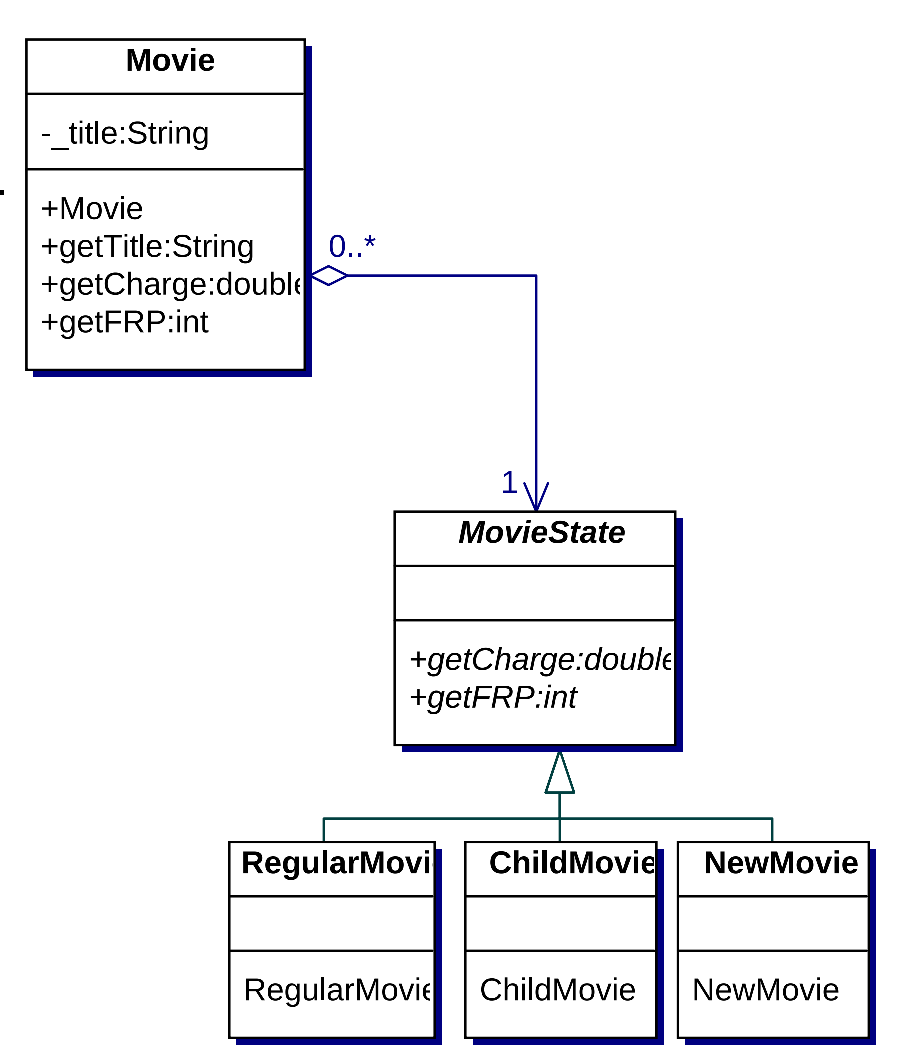
- **Name**: State
- **Absicht**: zustandsabhängige komplexe Berechnungen (insbesondere mit switch-Statements) werden in eigene Klassen ausgelagert.
- **Motivation**: oft besitzen Objekt eine kleine Anzahl an Zuständen, die auf ihr Verhalten großen Einfluss haben. Werden bei einem Objekt durch Zustandsänderungen die Verhaltensweisen mehrere komplexer Methoden geändert, so empfiehlt sich die Erzeugung einer Zustandsklasse je Objektzustand und die Auslagerung der Berechnungen als redefinierbare Methoden in die Zustandsklassen.
- ...
- **Beispiel**: Movie mit Berechnung von Charge und F(requent)R(enter)P(oints) aus dem vorigen Abschnitt.



Beispiel für „State“-Entwurfsmuster:

Die Klasse MovieState besitzt zwei abstrakte Methoden getCharge und get-FRP, die in den drei Unterklassen mit den speziellen Berechnungsvorschriften redefiniert werden.

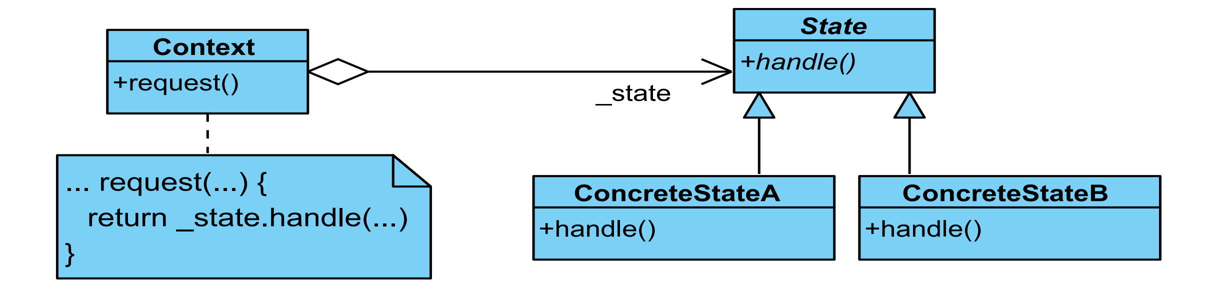
Die Assoziation zwischen Movie und MovieState zeigt bereits, dass es geplant ist, ein MovieState-Objekt in mehreren Movie-Objekten zu verwenden (anstatt viele gleiche MovieState-Objekte zu erzeigen) - siehe Flyweight-Pattern.

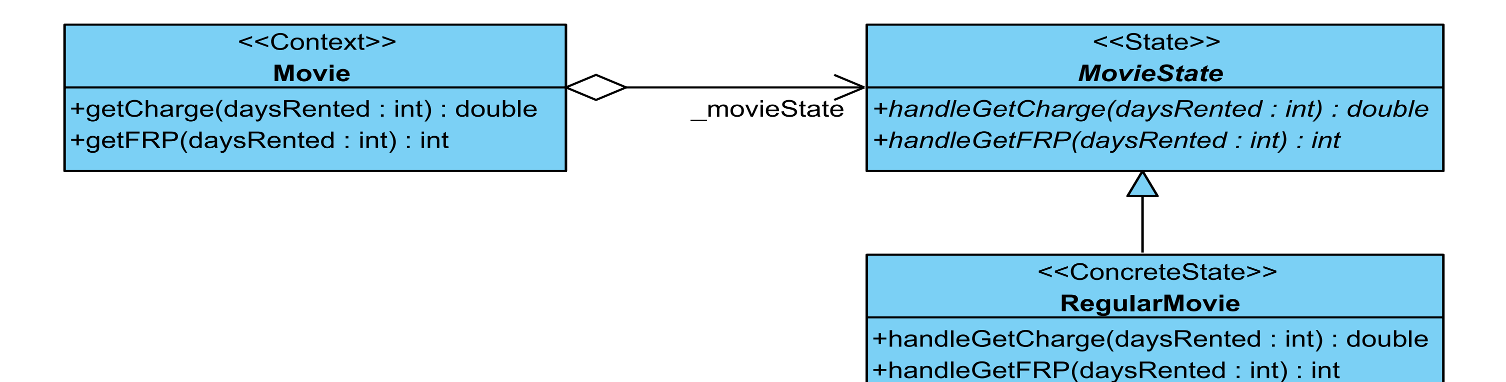




Genauere Definition des „State“-Entwurfmusters und -Anwendung:









Das Entwurfsmuster „Singleton“ (verkürzte Beschreibung):

- **Name**: Singleton
- **Absicht**: Realisierung einer Klasse mit genau einer Instanz.
- **Motivation**: anstelle von Klassen mit einer Instanz werden ggf. nicht instanziierbare Klassen mit statischen Attributen und Methoden verwendet. Oft benötigt man jedoch aus technischen Gründen (Vererbung, Übergabe der Objekte als Parameter) Objekte solcher Klassen. Die Implementierung muss aber sicherstellen, dass es nur genau eine Instanz der entsprechenden Klasse gibt.

Singleton
-_instance:Singleton
#Singleton +getInstance:Singleto

```
Singleton _instance = null;
...
Singleton getInstance () {
  if (_instance == null) {
  _instance = new Singleton
  };
  return _instance;
}
```

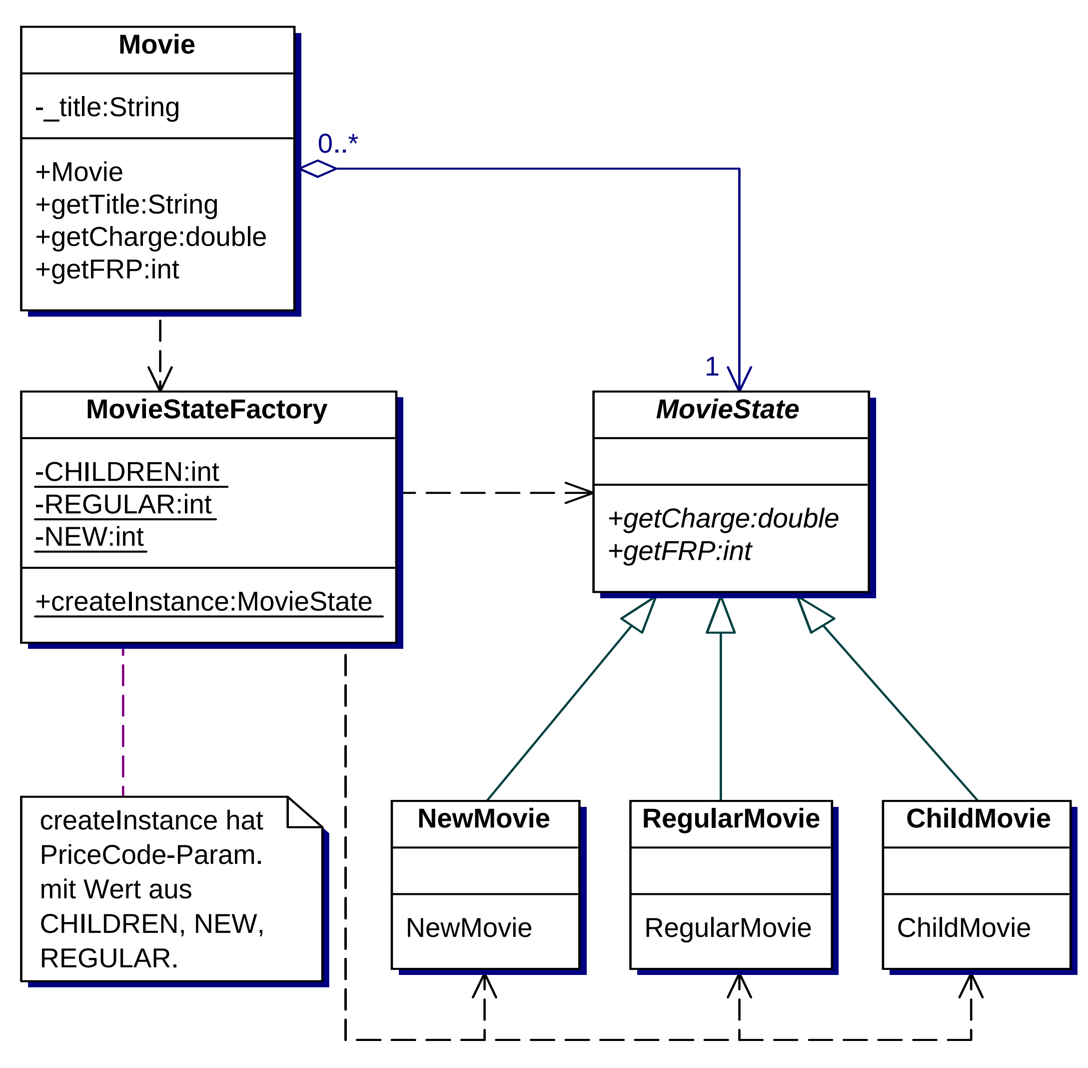


Das Entwurfsmuster „Factory“ (verkürzte Beschreibung):

- **Name**: Factory
- **Absicht**: eine Factory verkapselt die Erzeugung von Objekten. Der Anwender ruft eine Operation der Factory zur Erzeugung von Objekten einer anderen Klasse auf, anstatt direkt den Konstruktor der anderen Klasse zu rufen.
- **Motivation**: oft soll bei der Objekterzeugung verborgen bleiben, zu welcher Unterklasse einer abstrakten Oberklasse ein Objekt gehört, um so
  - ⇨ einen Algorithmus für die Auswahl der Unterklasse in einer eigenen Klasse zu „verstecken“ (und damit austauschbar zu machen)
  - ⇨ die Hinzunahme oder das Löschen neuer Unterklassen zu erleichtern
  - ⇨ technische Probleme bei der Verteilung von Programmen über mehrere Prozesse hinweg zu vermeiden
- **Beispiel**: in Abhängigkeit vom übergebenen Parameter priceCode wird beim Aufruf der Methode createInstance von MovieStateFactory entweder ein Objekt der Klasse NewMovie oder RegularMovie oder ChildMovie erzeugt



Beispiel für Factory-Muster:





Erläuterungen zu MovieStateFactory:

- Die Methode createInstance hat den im Diagramm nicht sichtbaren Parameter int priceCode, der die Werte CHILDREN, NEW und REGULAR annehmen darf.
- Von der MovieStateFactory wird kein Objekt angelegt, da alle Attribute und Methoden statisch sind.
- createInstance für eine bestimmte MovieState-Art überprüft zunächst, ob das entsprechende statische Attribut instance... bereits auf ein Objekt verweist. Falls ja, wird dieses Objekt zurückgeliefert.
- Ansonsten wird genau einmal ein neues ...Movie-Objekt erzeugt, in dem statischen Attribut instance... abgespeichert und zurückgeliefert.
- Alternativ kann man bereits vor dem ersten Aufruf von createInstance die statischen Attribute mit den entsprechenden Objekten initialisieren
- Statt drei verschiedene Attribute anzulegen, sollte man vielleicht besser ein Array (einen Vector) von MovieState-Objekten verwenden.



Das Entwurfsmuster „Abstract Factory“:

- **Name**: Abstract Factory
- **Absicht**: eine Abstract Factory führt eine weitere Abstraktionsstufe für die Erzeugung von Objekten ein. Der Auswahlprozess erfolgt zweistufig: zunächst wird eine konkrete Factory ausgewählt, die dann bestimmte Objektinstanzen erzeugt.
- **Motivation**: bei der Objekterzeugung soll nicht nur verborgen bleiben, zu welcher Unterklasse einer abstrakten Oberklasse ein Objekt gehört, sondern darüber hinaus soll der Algorithmus zur Auswahl einer Unterklasse zur Laufzeit austauschbar sein. Dies geschieht durch die Verwendung einer anderen konkreten Factory.
- **Beispiel**: eine Implementierung des Videoverleihsystems soll zur Laufzeit änderbare Schemata für die Preisgestaltung und Berechnung von Bonuspunkten haben. Für jedes Preisschema wird eine konkrete Factory realisiert, die dann die dazu passenden MovieState-Klassen erzeugt.



Struktur des „Abstract Factory“-Musters:

Design Pattern Abstract Factory:

AbstractFactory
+createProductA()
+createProductB()

Client

AbstractProductA

AbstractProductB

ConcreteFactory2
+createProductA()
+createProductB()

'ConcreteProductB2

ConcreteProductA2

ConcreteFactory1
+createProductA()
+createProductB()

ConcreteProductA1

ConcreteProductB1



Erläuterungen zum „Abstract Factory“-Muster:

- Ein Client bekommt eine Instanz der Klasse AbstractFactory übergeben, die entweder zur Unterklasse ConcreteFactory1oder ConcreteFactory2 gehört (Erzeugung einer Instanz einer dieser beiden Unterklassen könnte wieder durch ein Factory-Entwurfsmuster geregelt werden).
- Wenn die Client-Instanz ein Objekt der Klasse AbstractProductA oder der Klasse AbstractProductB benötigt, ruft sie die entsprechenden Methoden der übergebenen AbstractFactory-Instanz auf.
- Die createProduct<x>-Methoden der ConcreteFactory<i> erzeugen dann Instanzen der Unterklasse ConcreteProduct<x><i>.
- Die Client-Instanz weiß aber nur, dass Instanzen der abstrakten Klassen AbstractProduct<x> erzeugt werden.
- So kann man mit Hilfe einer „Abstract Factory“ die Algorithmen zur Auswahl der Klasse zu erzeugender Objekte für mehrere abstrakte Produktklassen gleichzeitig auswählen und umschalten.

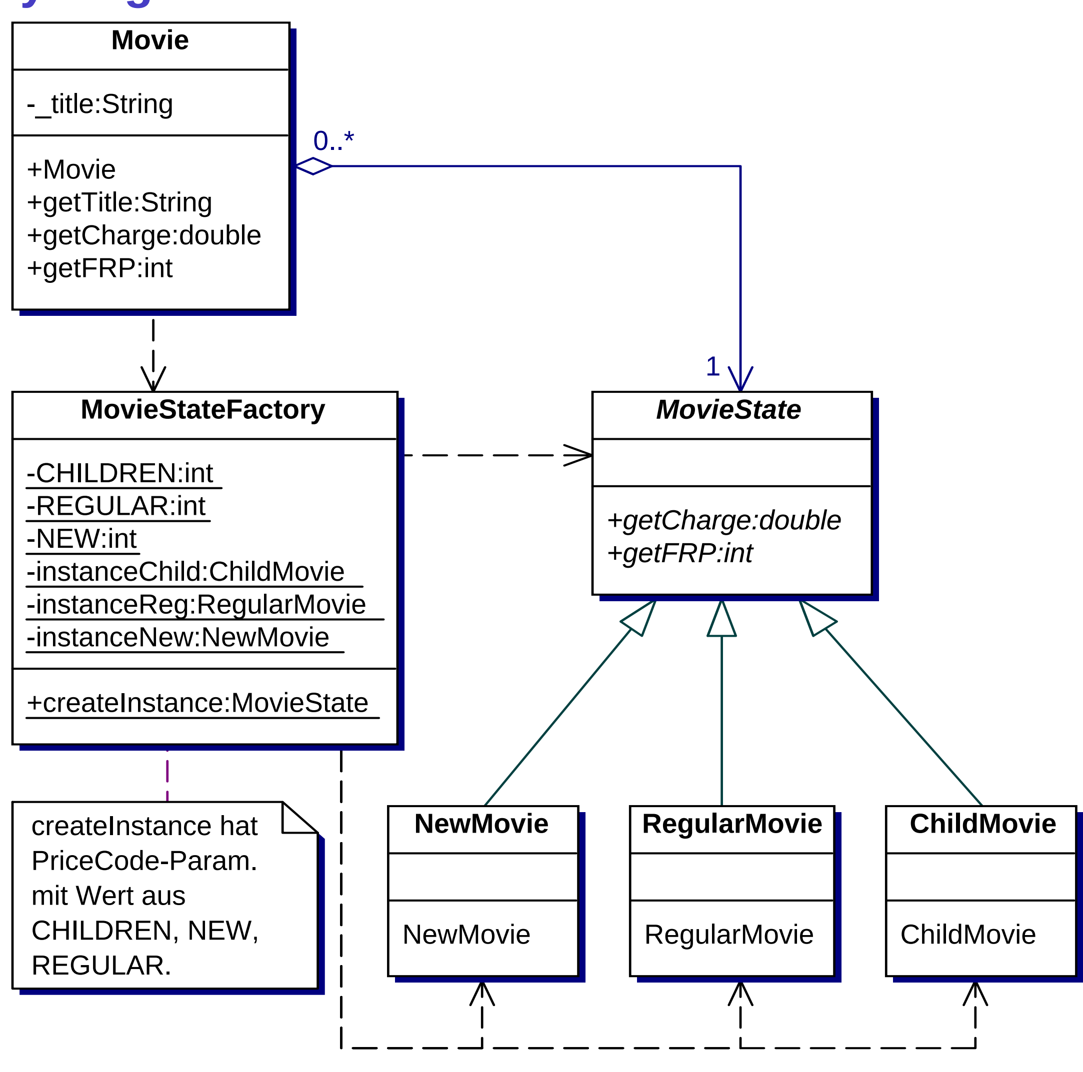


Das Entwurfsmuster „Flyweight“ (verkürzte Beschreibung):

- **Name**: Flyweight (Fliegengewicht)
- **Absicht**: benötigt man sehr viele (kleine) und immer identische Objekte, so erzeugt man diese nur einmal, speichert sie an geeigneter Stelle (in einer Factory) und verwendet sie immer wieder.
- **Motivation**: oft werden in einem Programm viele (kleine) Objekte angelegt, die alle denselben Zustand besitzen. Mit dem Flyweight-Pattern erzeugt man je benötigtem Zustand nur ein Objekt und verwendet diese immer wieder (shared objects). Man spart sich so die Laufzeit für die Erzeugung und Löschung der Objekte sowie den Speicherplatz für die sonst angelegten Duplikate.
- **Beispiel**: die für die PriceCode-Behandlung eingeführten MovieState-Objekte besitzen alle keinen eigenen Zustand. Je Unterklasse muss also nur genau ein Objekt angelegt werden, das von der MovieStateFactory verwaltet wird.

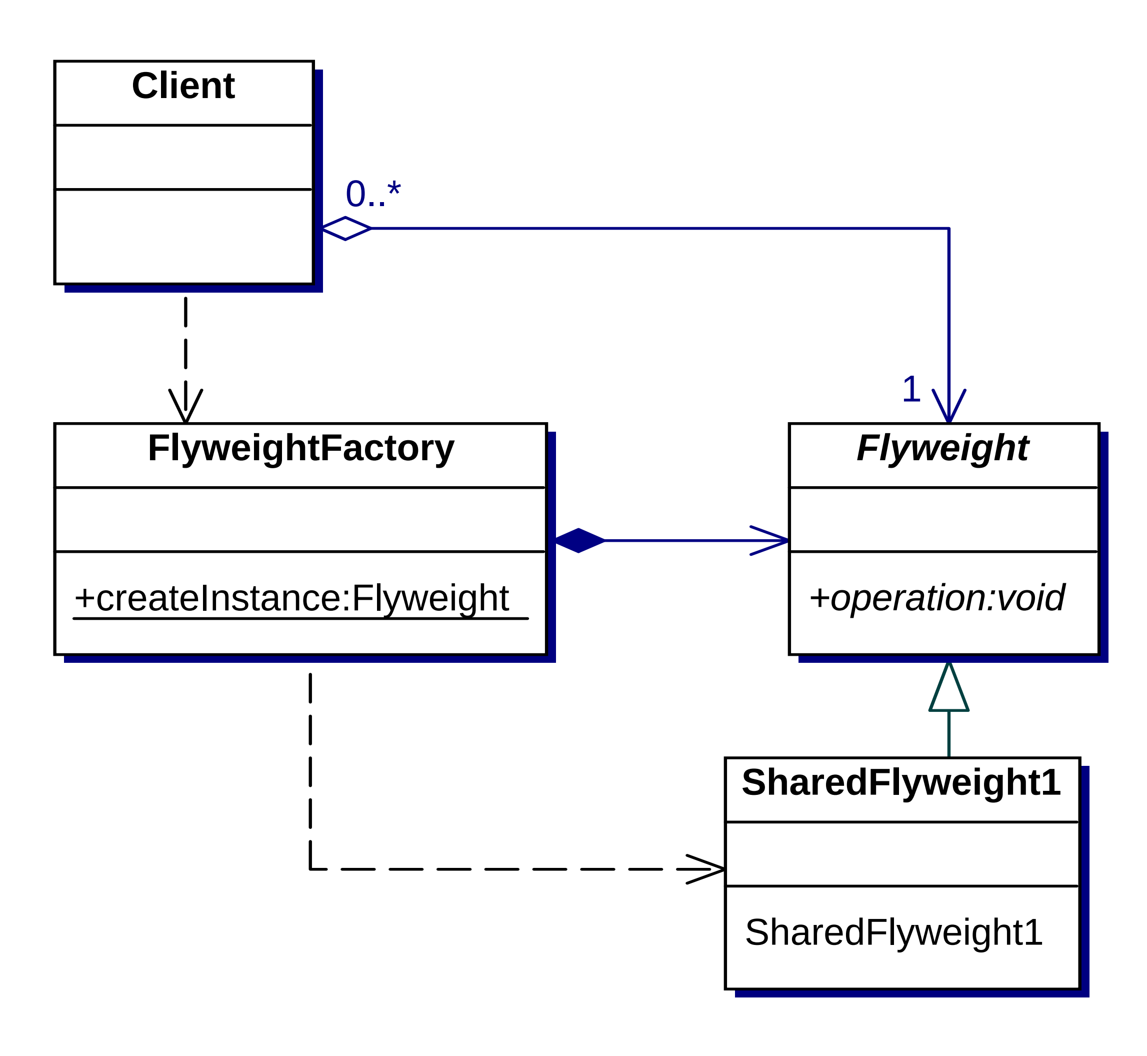


Beispiel für Flyweight-Muster:





Struktur des „Flyweight“-Musters:





Implementierungshinweise zu Flyweight:

- die Kompositionsbeziehung von FlyweightFactory zur abstrakten Klasse Flyweight deutet an, dass ein Vector/Array oder eine andere Datenstruktur zur Speicherung (Cashing) von erzeugten Flyweight-Instanzen benutzt wird
- die Unterklasse SharedFlyweight1 steht repräsentativ für viele Unterklassen der abstrakten Klasse Flyweight; nicht alle Unterklassen müssen am „Sharing“ unbedingt teilnehmen
- nur im einfachsten Fall gibt es von jeder Flyweight-Unterklasse genau eine Instanz; oft gibt es (unveränderliche) Attribute mit kleinem Wertebereich und es wird je Attributwert eine Instanz angelegt und in der FlyweightFactory vermerkt
- Client-Objekte erzeugen nie selbst Flyweight-Instanzen, sondern machen das immer über die FlyweightFactory
- es ist ein Implementierungsgeheimnis der FlyweightFactory, welche Objekte neu erzeugt werden und welche öfter verwendet und daher von mehreren Client-Objekten verwendet werden (sharing)



Das Entwurfsmuster „Chain of Responsibility“ (verkürzte Beschreibung):

- **Name**: Chain of Responsibility
- **Absicht**: die Durchführung einer Berechnung wird (teilweise) an eine Komponente der betrachteten Klasse delegiert.
- **Motivation**: auf diese Weise lassen sich komplexe Berechnungen in eigene leichter änderbare Klassen auslagern und der Aufrufer der Berechnung wird von der Durchführung der Berechnung „entkoppelt“ (Zwischenstationen können zusätzliche Teilberechnungen einführen).
- ...
- **Beispiel**: get(Total)Charge und get(Total)F(requent)R(enter)P(oints)
- **Implementierung**: die Delegation von Aufrufen an vorhandene Klassen wird oft beispielsweise anstelle der Vererbung und Redefinition von Methoden als besseres Entwurfsmittel eingesetzt; aber übermäßige Verwendung langer Verantwortlichkeitsketten führt zu ineffizienter und schwer zu ändernder Software (siehe zu eliminierende **„middleman“-Klassen** beim Refactoring)



„Chains of Responsibility" im Beispiel:

Movie
-_title:String
+Movie
+getTitle:String
+getCharge:double
+getFRP:int

RentalComponent
-_dateRange:String
+RentalComponent
+getDateRange:String
+getCharge:double
+getFRP:int

VisitedElement
Customer
-_name:String
+Customer
+getRental:RentalComponent
+getName:String
+printStatement:String
+htmlStatement:String
-getCharge:double
-getFRP:int

<<Leaf>>
RentalLeaf
-_movie:Movie
-_daysRented:int
-_charge:double
-_frequentRenterPoints:int
+RentalLeaf
+getMovie:Movie
+getCharge:double
+getFRP:int

RentalComposite
+RentalComposite
+getCharge:double
+getFRP:int
+addChild:void
+iterate:RentalIterator
first:Index
next:Index
current:RentalIterator

RentalIterator
-_current:Index
+RentalIterator
+first:void
+next:void
+current:RentalComponent
+isDone:boolean

MovieState
+getCharge:double
+getFRP:int

Videoshop
-_instance:Videoshop
#Videoshop
+getInstance:Videoshop

1 1 1 0..* 0..* 1 1 0..*

Index _current ist entweder Zahl/Index für Ve
oder Zeiger auf nächste RentalComponent

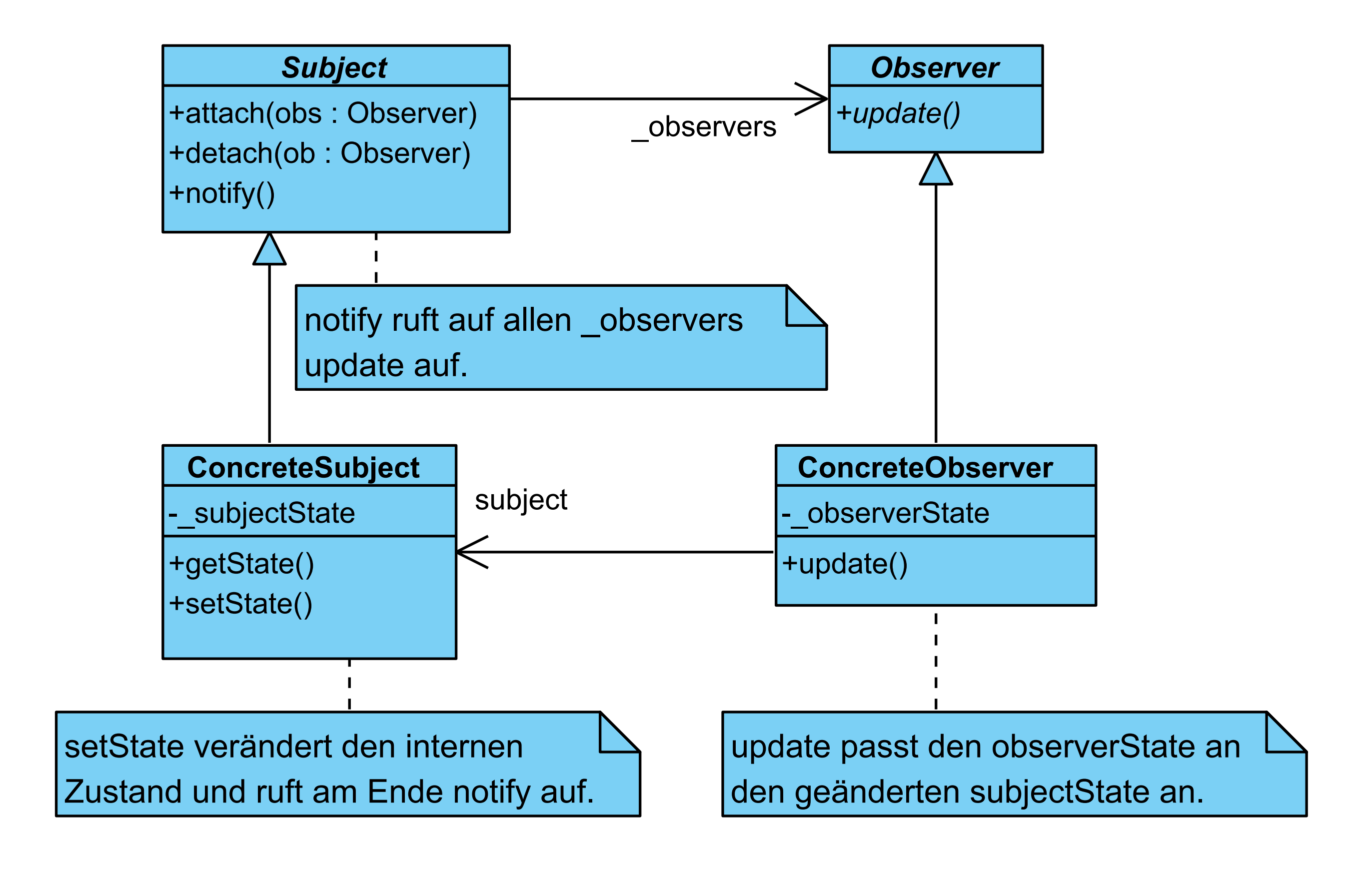


Das letzte Entwurfsmuster - „Observer“:

- **Name**: Observer (Beobachter)
- **Absicht**: etabliert eine 1-zu-n-Beziehung zwischen einem Subjekt und einer Menge von Beobachtern, die über Änderungen des Subjekts informiert werden.
- **Motivation**: erlaubt die „saubere“ Trennung von Logik (Model) und Darstellung (View) in einem Programm. Das Objekt Subject (Model) informiert alle seine Observer (Views) über jede Zustandsänderung, damit sich diese die Darstellung aktualisieren können.
- **Beispiel**: Die Klasse Customer unseres Videoverleihprogramms bietet immer noch eine bunte Mischung von Methoden an, die einerseits logische Berechnungen durchführen (getName, ...) und andererseits Darstellungen berechnen (print-Statement, ...). Das Observer-Entwurfsmuster erlaubt uns die Auftrennung der alten Klasse Customer in CustomerSubject und CustomerHtmlObserver etc.
- **Anmerkung**: Die Control-Klasse des MVC-Konzepts fehlt hier noch, die View-/Obeserver-Objekte bei Model-/Subject-Objekten registriert.



Klassendiagramm des Entwurfsmusters „Observer“:

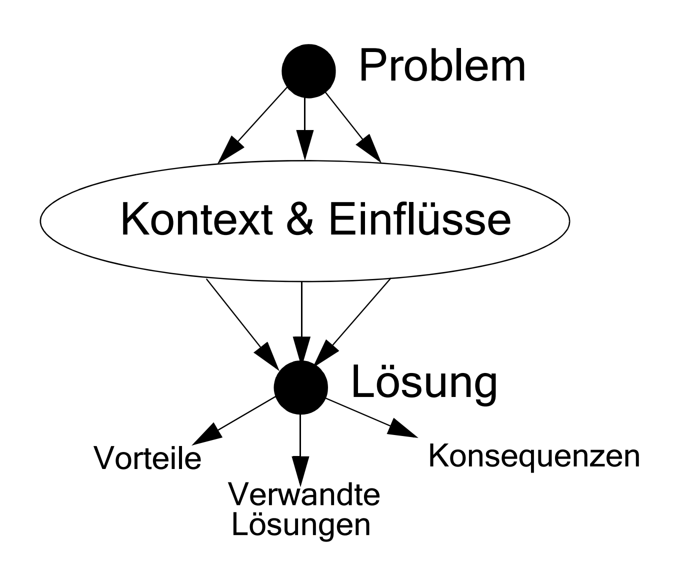




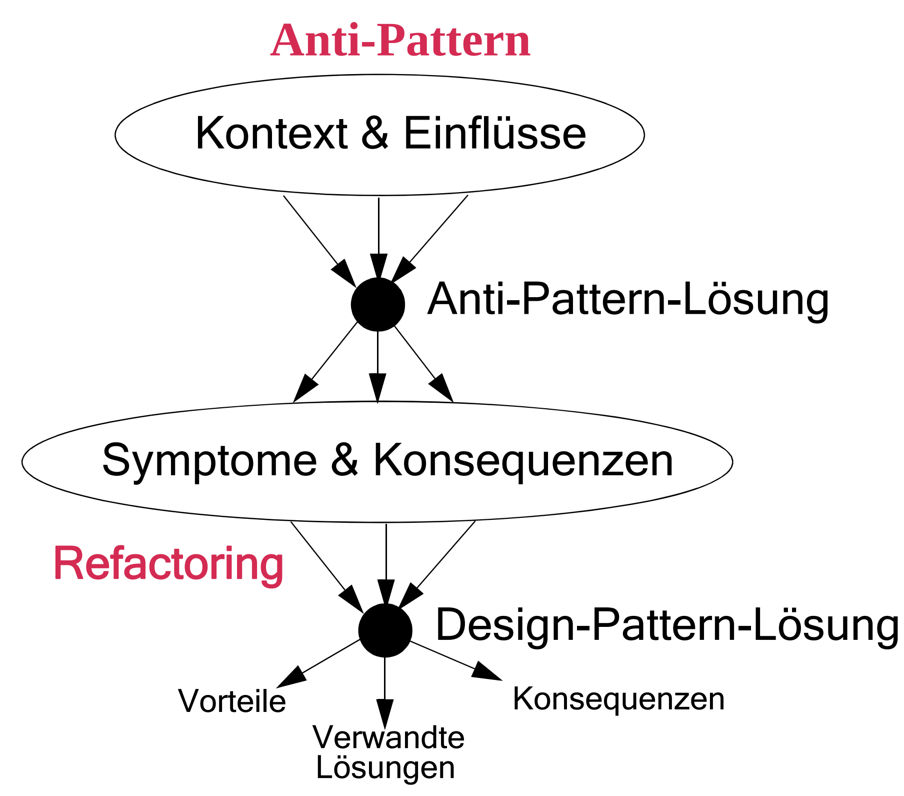
Von Design-Pattern zu Anti-Pattern:

Anti-Pattern beschreiben häufig verwendete Lösungen (Entwurfsmuster) für Probleme, die sich in der Praxis nicht bewähren (negative Konsequenzen haben). Sie sind damit das Gegenstück zu den Design-Patterns, die bewährte Lösungen für oft wiederkehrende Probleme festhalten.

(Design-)Pattern



Anti-Pattern





Definition von Anti-Pattern:

In Anlehnung an [BM+98] benutzen wir folgendes Schema für die Definition von Anti-(Design-)Pattern bzw. Anti-Entwurfsmuster, das allerdings etwas verkürzt und an unser Schema für Entwurfsmuster angepasst wurde:

- **Name + Synonyme**: Namen unter denen das Anti-Muster bekannt ist
- **Hintergründe**: Beschreibung der Hintergründe der Verwendung des Anti-Musters
- **Struktur/Form**: Beschreibung des Aufbaus des Anti-Musters (zur Identifikation)
- **Konsequenzen**: Beschreibung der negativen Auswirkungen, die sich aus der Existenz des Anti-Musters ergeben
- **Gründe**: Beschreibung der typischen Gründe für die Existenz des Anti-Musters
- **Beispiel**: ein Beispiel, das den Aufbau des Anti-Musters verdeutlicht
- **Neue Lösung**: Hinweise zum Refactoring, i.e. zur Ersetzung des Anti-Musters durch ein empfehlenswertes Entwurfsmuster
- **Erlaubte Ausnahmen**: Umstände, unter denen das Anti-Muster akzeptabel ist und kein Refactoring durchgeführt werden soll/muss



Das Anti-Entwurfsmuster „Blob“:

- **Name**: Blob (Schleimmonster)
- **Synonyme**: „The God Class“, ...
- **Hintergründe**: Erinnern Sie sich an den Film „The Blob“? Ein außerirdisches Schleimmonster trifft auf der Erde ein, frisst alles auf und wächst und wächst ... . Der Film ist ein gutes Beispiel für eine Klasse, die immer mehr Verantwortlichkeiten und Code in einem System an sich zieht.
- **Struktur/Form**: eine Klasse mit einer sehr großen Anzahl an Methoden und Attributen, die hauptsächlich nur Daten verwaltende andere Klassen benutzt. Eine oder mehrere Methoden sind typischerweise selbst wiederum sehr groß und komplex.
- **Konsequenzen**: Änderungen unterschiedlicher logischer Sachverhalte müssen immer wieder in der selben Klasse durchgeführt werden; diese Blob-Klasse ist zudem kaum wiederverwendbar und sehr schwer zu testen.
- **Gründe**: von der Verwendung „prozeduraler“ Programmiersprachen geprägte Entwickler neigen dazu, insbesondere bei der Realisierung eingebetteter Systeme die Kontroll-Logik des Systems in einer Klasse (und einer Methode) zu verankern.



Das Anti-Entwurfsmuster „Blob“ - Fortsetzung:

- **Beispiel**: die ursprüngliche Klasse Customer unserer Videoverleihsoftware mit ihren großen Methoden (print)statement und htmlStatement.
- **Neue Lösung**: Auslagerung von logisch zusammengehörigen Attributen und Methoden in eigene Klassen (oder bereits existierende Klassen), Zerlegung von Berechnungen durch Delegation (Chains of Responsibilities) in Teilberechnungen, Einführung von Oberklassen, die erweiterbares Grundverhalten einführen, ...
- **Erlaubte Ausnahmen**:
  - ⇨ Verkapselung von „Altlasten“, deren Refactoring im Augenblick nicht möglich ist oder sich nicht (mehr) lohnt, in einer Klasse
  - ⇨ [Entwickler, die sich nicht davon überzeugen lasssen, dass die Zerlegung einer großen Klassen in viele kleine Klassen zu einem übersichtlicheren und (fast) genauso effizienten Entwurf führt]



Das Anti-Entwurfsmuster „Poltergeist“:

- **Name**: Poltergeist
- **Synonyme**: „Big Do It Controller Class“, ...
- **Hintergründe**: Poltergeister sind unerwünschte Zeitgenossen, die kurze Zeit zu sehen/hören und dann wieder verschwunden sind. Das gleiche gilt für die Objekte von Poltergeist-Klassen, die erzeugt und kurz darauf wieder gelöscht werden.
- **Struktur/Form**: eine Klasse mit begrenzten Aufgaben. Typischerweise kontrolliert sie das Zusammenspiel anderer Klassen für einen kurzen Zeitraum (Abarbeitung einer Berechnung) und besitzt kaum eigene Attribute. Typische Namen für solche Klassen sind „...Manager“ oder „...Controller“.
- **Konsequenzen**: das permanente Erzeugen und Löschen der Objekte solcher Klasen führt zu ineffizienten Programmen. Zudem erschweren sie Änderungen an den von ihnen kontrollierten Klassen.
- **Gründe**: von der Verwendung „prozeduraler“ Programmiersprachen geprägte Entwickler neigen dazu, insbesondere bei der Realisierung eingebetteter Systeme die Kontroll-Logik des Systems in einer Klasse (und einer Methode) zu verankern.



Das Anti-Entwurfsmuster „Poltergeist“ - Fortsetzung:

- **Beispiel**: unser Videoverleihsystem enthält keine solche Klasse.
- **Neue Lösung**: die Klasse wird gelöscht und ihr Verhalten auf meist bereits existierende bislang von ihr kontrollierte Klassen verteilt. Falls das nicht möglich ist, kann man zumindest das dauernde Erzeugen und Löschen der Poltergeister durch Einsatz des „Flyweight“-Patterns vermeiden.
- **Erlaubte Ausnahmen**:
  - ⇨ keine - sagt [BM+98]
  - ⇨ Koordinationsklassen, deren Instanzen nicht dauernd erzeugt und gelöscht werden und die eigene Attribute besitzen (und keine Schleimmonster sind)
  - ⇨ Kapselung von Algorithmen zum Zwecke der Wiederverwendung, Austauschbarkeit und vor allem Übergabe als Parameter an Methoden



Abschließende Anmerkungen zu Anti-Patterns:

- in [BM+98] findet man noch eine ganze Reihe weiterer Anti-Patterns
- bislang gibt es Dutzende von Büchern über Design-Patterns wie z.B. [BMR96], [GH+94], aber es gibt nur sehr wenige Bücher über Anti-Pattern
- zudem befassen sich die meisten Anti-Pattern-Bücher nicht nur (gar nicht) mit Anti-Patterns, sondern mit „schädlichen“ Praktiken des Projektmanagements, Konfigurationmanagements, ...
- allerdings befassen sich auch Refactoring-Bücher mit unerwünschten Entwurfs-mustern, die ja durch systematische Umbauten eliminiert werden
- ein interessantes Buch über Anti-Patterns beim Programmieren in Java ist [Ta02]

Fazit:

Auf diesem Gebiet wird sich in den nächsten Jahren noch einiges tun, auch wenn die Niederschrift „schlechter“ Entwurfsmuster erfahrungsgemäß schwieriger und weniger populär ist als die Veröffentlichung „guter“ Entwurfsmuster.



Grenzen der Bewertung von Softwaregüte durch Anti-Pattern:

Pattern und Anti-Pattern betrachten immer die Güte von Software **lokal**, globale Eigenschaften einer Softwarearchitektur werden nicht berücksichtigt.

Beispiele für globale „Bad Smells“ einer Softwarearchitektur:

- **zyklische Benutzt-Beziehungen** (Importe) über Teilsysteme (Pakete) hinweg
  - ⇨ wechselseitige direkte oder indirekte Benutzungen führen dazu, dass die betroffenen Teilsysteme (Pakete) nicht mehr unabhängig austauschbar sind
  - ⇨ Reparaturen erfordern oft größere Refactoring-Maßnahmen
- **Verletzungen einer Schichtenarchitektur**:
  - ⇨ Pakete/Klassen werden Schichten eines Softwaresystems zugeordnet
  - ⇨ Pakete/Klassen einer Schicht dürfen immer nur Pakete/Klassen der (direkt) darunterliegenden Schichten benutzen
  - ⇨ Verletzungen findet man in (fast) allen großen Softwaresystemen
  - ⇨ Reparaturen erfordern oft ebenfalls größere Refactoring-Maßnahmen



8.6 Zusammenfassung

Der systematische Umbau und die Bewertung von Software wurde in diesem Kapitel vertieft. Für den Alltag der Softwareentwicklung sollte man sich folgendes merken:

- Der Begriff „**Softwarearchitektur**“ wird in der Literatur und Praxis sehr unterschiedlich definiert (siehe hier aufgezählte Architektursichten). Oft sind strukturelle Sichten gemeint wie z.B. UML-Klassen und Paket-Diagramme
- Achtung: bei **Architekturbeschreibungssprachen** wird oft gefordert, dass deren Basiskomponenten Schnittstellen anbieten und fordern können und dass Schnittstellen-Konnektoren eigenständige Implementierungsobjekte sind
- Für die Abbildung von Software auf Hardware können in UML **Einsatzdiagramme** verwendet werden (werden aber nicht so oft bislang genutzt)
- **Pattern und Anti-Pattern** sind das Mittel der Wahl für die Dokumentation und Weitergabe von Erfahrungswissen über gute und schlechte Softwareentwürfe.
- **Refactoring** ist eine wichtige Technik zur systematischen Sanierung schlecht strukturierter Software; dabei wird die Gefahr des Einbaus von (neuen) Fehlern minimiert (durch verhaltensbewahrende kleine Transformationsschritte)



8.7 Weitere Literatur

[Al77] Alexander Ch. : *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press (1977) 1216 Seiten

Die Inspirationsquelle aus dem Fachgebiet Architektur, auf die sich die Software-Muster-Gemeinde bezieht; es geht um Architekturstile und vorgefertige Teillösungen für den Entwurf von Gebäuden

[BCK98] Bass L., Clements P., Kazman R.: *Software Architecture in Practice*, Addison Wesley (1998), 452 Seiten

Von Praktikern geschriebenes Buch, das anhand verschiedenster Beispiele aus der "realen" Welt das Thema Softwarearchitekturen ausgiebig diskutiert. Leider spielt Objektorientierung kaum eine Rolle und UML wird nicht einmal erwähnt.

[BM+98] Brown W.H., Malveau R.C., McCormick III H.W. , Mowbray Th.J. : *Anti Patterns - Refactoring Software, Architectures and Projecst in Crisis*, John Wiley & Sons (1998), 309 Seiten

Eines der wenigen Bücher bislang zum Thema Anti-Pattern, also erwiesenermaßen ungeeignete Software-Entwurfsmuster.

[BMR96] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: *Pattern-orientierte Softwarearchitektur*, Addison Wesley (1996)

Ein Standardwerk zum Thema Softwareentwurf mit Mustern. Im Gegensatz zu [GH+94] werden hier nicht nur Entwurfsmuster für kleine Teile einer Softwarearchitektur beschrieben, sondern auch Modellierungs-stile, die eine Softwarearchitektur als Ganzes betreffen (Schichtenarchitektur, …).



[Fo97] Fowler M.: *Analysis Patterns*, Addison Wesley (1997), 357 Seiten

Ein erster Versuch, die Prinzipien der Softwareentwicklung mit Mustern (patterns) von der Entwurfs- auf die Analysephase zu übertragen

[Fo00] Fowler M.: Refactoring - Wie Sie das Design vorhandener Software verbessern, Addison Wesley (2000)

Das Standard-Buch zum Thema „Refactoring“, auf das dieses Kapitel aufbaut

[GH+94] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison Wesley (1994)

Das Standardbuch — neben [BMR96] — das den Entwurf von Softwarearchitekturen mit einer Familie von Standardentwurfsmustern (design patterns) vorstellt.

[La98] Larman C.: *Applying UML and Patterns*, Prentice Hall (1998)

Eines der ersten UML-Bücher, das anhand eines durchgängigen Beispiels ein Vorgehensmodell zum Einsatz von UML vorstellt. Eine vereinfachte und abgeänderte Version dieses Vorgehensmodells wird in dieser Vorlesung benutzt.

[RL04] Roock St., Lippert M.: *Refactorings in großen Softwareprojekten*, dpunkt.verlag (2004), 272 Seiten

„Nomen est omen“. Es geht in diesem Buch um die Sanierung umfangreicher Softwaresysteme anstelle der hier vorgestellten lokalen Optimierungsmaßnahmen

[Ta02] Tate B.A.: *Bitter Java*, Manning Publications (2002), 350 Seiten

Buch zum Thema „Bad Smells“ bei der Java-Programmierung



9. Qualitätssicherung und Testverfahren

Themen dieses Kapitels:

- Übersicht über Softwarequalitätssicherungsmaßnahmen
- Fehlersuche als ein Mittel (unter vielen) zur Qualitätsverbesserung
- systematische Verfahren zur Fehlersuche
- Einfluss von UML-Modellen auf die Fehlersuche

Achtung:

Dieses Kapitel lehnt sich an Teil III von [Ba98] und an Teil 5 von [So01] an, gibt aber nur eine kurze Zusammenfassung der dort angesprochenen Themen. Weiteres zum Thema „Testen“ als ein Mittel der Qualititätssicherung in der Vorlesung „Software Engineering II“. Für umfassende Informationen zum Thema „Testen (objektorientierter Systeme)“ wird auch [Bi99] empfohlen.



9.1 Softwarequalitätssicherung im Überblick

Zur Erinnerung:

Das Ziel der Software-Technik ist die Bereitstellung von Methoden, Sprachen und Werkzeugen zur effizienten Entwicklung messbar **qualitativ hochwertiger Software**, die

- ⇨ korrekt bzw. zuverlässig arbeitet
- ⇨ vertrauenswürdig ist
- ⇨ robust auf Fehleingaben, Hardwareausfall, Angriffe, … reagiert
- ⇨ effizient (ökonomisch) mit Hardwareressourcen umgeht
- ⇨ benutzerfreundliche Oberfläche besitzt
- ⇨ …

Achtung:

Im Folgenden befassen wir uns vor allem mit Maßnahmen zur **Erhöhung der Zuverlässigkeit** von Software.



Zur Erinnerung - Definitionen einiger Qualitätsmerkmale:

- **zuverlässige Software** funktioniert meistens; Zuverlässigkeit ist also ein Maß für die Wahrscheinlichkeit,
  - ⇨ dass ein Software-System sich in einem bestimmten Zeitraum so verhält, wie es von ihm erwartet wird
  - ⇨ oder dass ein Software-System einen angeforderten Dienst wie erwartet ausführt
- **korrekte Software** ist Software, die sich immer so verhält, wie in den Anforderungen festgelegt wurde; ob sie sich damit so verhält, wie man (der Anwender) das erwartet, bleibt damit offen (korrekte Software muss also nicht zuverlässig sein)
- **vertrauenswürdige Software** verursacht niemals Katastrophen; also auch dann nicht, wenn sie sich nicht korrekt verhält
- **robuste Software** funktioniert bzw. reagiert auch unter unvorhergesehenen Umständen „vernünftig“ (z.B. auf unerwartete Fehleingaben oder zufällige bzw. beabsichtige Angriffe)



Zur Erinnerung - Metriken für Bewertung der Zuverlässigkeit:

1. **„rate of failure occurrence" (ROFOC)**: Häufigkeit von nicht erwartetem Verhalten, z.B. „bei der Benutzung des Fahrzeugreservierungssystems (MVRS) treten pro Monat zwei Fehlfunktionen auf"

2. **„mean time to failure" (MTTF)**: mittlerer Zeitabstand zwischen zwei Fehlern, z.B. „Fahrzeugreservierungssystem funktioniert im Mittel zwei Wochen fehlerfrei"

3. **„availability" (AVAIL)**: mittlere Verfügbarkeit der Software, z.B. „an 2 von 1000 Arbeitsstunden ist das Fahrzeugreservierungssystem (wegen Fehlerbehebungs- oder Wartungsmaßnahmen oder …) nicht benutzbar"

Eine neue Metrik:

4. **„probability of failure on demand" (POFOD)**: Wahrscheinlichkeit der fehlerhaften Ausführung (mit unerwartetem bzw. nicht erwünschtem Ergebnis) eines angeforderten Dienstes, z.B. „jede tausendste Fahrzeugreservierung wird falsch durchgeführt"



Verlässlichkeit - ein neuer Begriff:

In [So01] wird „**Verlässlichkeit**“ eines (Software-)Systems als umfassender und systematischer definiertes Qualitätsmerkmal eingeführt.

Die Verlässlichkeit eines Systems ergibt sich aus der Kombination folgender Eigenschaften:

1. **Verfügbarkeit**: Fähigkeit des Systems, Dienste auf Anforderung zu liefern (charakterisierbar durch AVAIL-Metrik)

2. **Zuverlässigkeit**: Fähigkeit des Systems, Dienste wie spezifiziert zu liefern (charakterisierbar durch POFOD-Metrik, …)

3. **Betriebssicherheit**: Fähigkeit des Systems ohne katastrophale Ausfälle zu operieren (bei uns „Vertrauenswürdigkeit“ genannt)

4. **Systemsicherheit**: Fähigkeit des Systems sich auch gegen zufällige oder beabsichtigte Angriffe zu schützen (lässt sich als Teilaspekt der „Robustheit“ eines Softwaresystems auffassen)



Qualititätsmanagement im Überblick:

- bislang vorgestellte Vorgehensweise zur Entwicklung von vornherein qualitativ hochwertiger Softwareprodukte
  - „**Fehlervermeidung**“ durch **konstruktive Qualititätssicherung**
- im Folgenden geht es um die nachträgliche Überprüfung der Qualität entwickelter Produkte oder Dokumente
  - „**Fehlerelimination**” durch **analytische Qualitätssicherung**
- Überprüfung und Verbesserung der Qualität des Softwareentwicklungsprozesses selbst wird im nachfolgenden Kapitel 9 behandelt:
  - **ISO 9000** Zertifizierung bestätigt geregelte Vorgehensweise (trifft nahezu keine Aussage über Qualitität der eingesetzten Vorgehensweisen)
  - **Capability Maturity Model** (CMM) unterscheidet verschiedene Reife-grade bei Softwareentwicklungsprozessen und schlägt Verbesserungsmaß-nahmen vor



Prinzipien der Qualititätssicherung:

- **Qualitätszielbestimmung**: Auftraggeber und Auftragnehmer legen vor Beginn der Softwareentwicklung gemeinsames Qualitätsziel für Softwaresystem mit nachprüfbarem Kriterienkatalog fest (als Vertragsbestandteil des Lastenheftes → Abnahmetests)
- **quantitative Qualitätssicherung**: Einsatz automatisch ermittelbarer Metriken zur Qualitätsbestimmung (objektivierbare, ingenieurmäßige Vorgehensweise)
- **konstruktive Qualitätssicherung**: Verwendung geeigneter Methoden, Sprachen und Werkzeuge (Sprachen mit vernünftiger Syntax, statischem Typkonzept, …)
- **integrierte, frühzeitige analytische Qualitätssicherung**: nicht nur fertiges Softwareprodukt testen, sondern alle erzeugten Dokumente wie Analyse- und Designmodelle sowie einzelne Softwarekomponenten
- **unabhängige Qualitätssicherung**: Entwicklungsprodukte werden durch eigenständige Qualitätssicherungsabteilung überprüft und abgenommen (verhindert u.a. Verzicht auf Testen zugunsten Einhaltung des Entwicklungszeitplans)



9.2 Maßnahmen zur Erstellung fehlerarmer Software

Begriffe „Failure, Fault, Error” im Englischen:

- **Failure** (Fehlerwirkung): es handelt sich um ein Fehlverhalten eines Programms, das während seiner Ausführung (tatsächlich) auftritt.
- **Fault/Defect/Bug** (Fehlerzustand): es handelt sich um eine fehlerhafte Stelle (Zeile) eines Programms, die ein Fehlverhalten auslösen kann.
- **Error**: es handelt sich um eine fehlerhafte Aktion (Irrtum), die zu einer fehlerhaften Programmstelle führt (oder Ausführung fehlerhafter Zeile).

Oder anders formuliert:

Fehler bei der Programmierung (errors) können zu Fehlern in einem Programm (faults) führen, die Fehler bei der Programmausführung (failure) bewirken.

Achtung:

Anstelle von „Failure“ wird oft von „Error“ (Fehlverhalten auslösende Aktion) gesprochen; deshalb spricht man oft von „error codes“ oder „error handler“.



Entstehung und Fortpflanzung von Fehlern (Faults):

korrekt/fehlerhaft bzgl. eigentlich benötigtem Programmverhalten

Phase				
Machbarkeitsstudie (grobe Anforderungen)	korrekte Anforderungen	fehlerhafte Anforderungen		
Anforderungsanalyse (Systemspezifikation)	korrekte Spezifikation	Spezifikations-fehler	induzierte Fehler aus Anforderungen	
Entwurf (Design)	korrekter Entwurf	Entwurfs-fehler	induzierte Fehler aus …	
Codierung	korrektes Programm	Programm-fehler	induzierte Fehler aus …	
Test und Integration	korrektes Programm	korrigierte Fehler	gefundene nicht korrigierte Fehler	unbekannte Fehler



Konstruktives Qualitätsmanagement zur Fehlervermeidung:

- **technische Maßnahmen**:
  - Sprachen (wie z.B. UML für Modellierung, Java für Programmierung)
  - Werkzeuge (UML-CASE-Tool wie Together oder …)
- **organisatorische Maßnahmen**:
  - Richtlinien (Gliederungsschema für Pflichtenheft, Programmierrichtlinien)
  - Standards (für verwendete Sprachen, Dokumentformate, Management)
  - Checklisten (wie z.B. „bei Ende einer Phase müssen folgende Dokumente vorliegen" oder „Softwareprodukt erfüllt alle Punkte des Lastenheftes")

Einhaltung von Richtlinien, Standards und Überprüfung von Checklisten kann durch Werkzeugeinsatz = technische Maßnahmen erleichtert (erzwungen) werden.



Analytisches Qualitätsmanagement zur Fehleridentifikation:

- **analysierende Verfahren**:

  der „Prüfling“ (Programm, Modell, Dokumentation) wird von Menschen oder Werkzeugen auf Vorhandensein/Abwesenheit von Eigenschaften untersucht

  - **Inspektion** (Review, Walkthrough): organisiertes „Durchlesen“ in Gruppe
  - **statische Analyse**: werkzeuggestützte Ermittlung von „Anomalien“
  - **(formale) Verifikation**: werkzeuggestützter Beweis von Eigenschaften

- **testende Verfahren**:

  der „Prüfling“ wird mit konkreten oder abstrakten Eingabewerten von Menschen oder Werkzeugen ausgeführt

  - **dynamischer Test** (Simulation): Ausführung mit konkreten Eingaben
  - **symbolischer Test**: Interpretation mit symbolischen Eingaben (die oft unendliche Mengen möglicher konkreter Eingaben repräsentieren)
  - **Schreibtischtest**: Ausführung mit Papier und Bleistift



Wann kann welches Verfahren eingesetzt werden:

- **konstruktives** Qualititätsmanagement reduziert menschliche Fehler (errors) und senkt damit a priori die Anzahl der Fehler (faults)
- **analytisches** Qualitätsmanagement erhöht die Fehlereliminationsrate, dient also der Entdeckung und Behebung von Fehlern (faults):
  - ⇨ **Softwareinspektion** lässt sich in allen Phasen der Softwareentwicklung zur Elimination von Fehlern (faults) einsetzen
  - ⇨ die **statische Analyse** erfordert Modelle mit präzise definierten Konsistenzbedingungen (z.B. UML-Diagramme) oder Code; sie liefert in aller Regel „nur“ Hinweise auf Fehler (faults)
  - ⇨ **(formale) Verifikation** erfordert ausführbaren Code (oder ausführbares Modell) und beweist die Abwesenheit von fehlerhaften Programmstellen (faults)
  - ⇨ **Testen** erfordert ausführbaren Code (oder ausführbares Modell) und findet Programmfehler (faults) durch Auslösen von Fehlfunktionen (failure)



Verschiedene Arten von Codierungsfehlern:

- **Berechnungsfehler**: Komponente berechnet falsche Funktion
  - ⇨ Rundungsfehler bei der Berechnung einer Formel
  - ⇨ …
- **Schnittstellenfehler**: Inkonsistenz (bezüglich erwarteter Funktionsweise) zwischen Aufrufsstelle und Deklaration
  - ⇨ Übergabe falscher Parameter, Vertauschen von Parametern
  - ⇨ Verletzung der Randbedingungen, unter denen aufgerufene Komponente funktioniert (z.B. „Division durch Null“)
- **Kontrollflussfehler**: Ausführung eines falschen Programmpfades
  - ⇨ Vertauschung von Anweisungen
  - ⇨ falsche Kontrollbedingung (z.B. „kleiner“ statt „kleiner gleich“), „off by one“: Schleife wird einmal zuwenig oder zu oft durchlaufen
- **Initialisierungsfehler**: falsche oder fehlende Initialisierung einer Variablen
  - ⇨ Variable wird auf einem vieler möglicher Programmpfade nicht initialisiert



- **Datenflussfehler**: falscher Zugriff auf Variablen und Datenstrukturen
  - ⇨ falsche Arrayindizierung
  - ⇨ Zuweisung an falsche Variable
  - ⇨ Zugriff auf Null-Pointer
  - ⇨ Zugriff auf bereits freigegebenes Objekt
- [**Zeitfehler**: gefordertes Zeitverhalten nicht eingehalten - hier nicht betrachtet]
  - ⇨ Implementierung ist nicht effizient genug
  - ⇨ wichtige Interrupts werden zu lange blockiert

Neue Fehlerart im Zuge der objektorientierten Programmierung:

- **Redefinitionsfehler**: geerbte Operation wird nicht semantikerhaltend redefiniert
  - ⇨ ein „Nutzer" der Oberklasse geht von Eigenschaften der aufgerufenen Operation aus, die Redefinition in Unterklasse nicht (mehr) erfüllt



Beispiel für fehlerhafte Prozedur:

```
PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)
VAR i: INTEGER;
BEGIN
   count := 0;
   WHILE s[i] # '.' DO
      IF s[i]= 'a' OR s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN
         count := count + 2;
      END;
      count := i+1;
END countVowels
   …
countVowels('to be . . . or not to be.', count);
```

Achtung:

Diese in Pseudo-Pascal geschriebene Prozedur wird im Folgenden als durchgängiges Beispiel verwendet.



9.3 Manuelle statische Testverfahren

Systematische Verfahren zur gemeinsamen „Durchsicht“ von Dokumenten (wie z.B. erstellte UML-Modelle, implementierten Klassen, …):

- ⇨ **Programminspektion**: stark formalisiertes Verfahren bei dem Dokument nach genau festgelegter Vorgehensweise durch Gutachterteam untersucht wird
- ⇨ **Review**: weniger formale manuelle Prüfmethode; weniger Aufwand als bei Programminspektion, ähnlicher Nutzen
- ⇨ **Walkthrough**: unstrukturierte Vorgehensweise; Autor des Dokuments liest vor, Gutachter stellen spontan Fragen
- ⇨ [**Pair Programming**: Programm wird von vornherein zu zweit erstellt]

Empirische Ergebnise zu Programminspektion:

- ⇨ Prüfaufwand liegt bei ca. 15 bis 20% des Erstellungsaufwandes
- ⇨ 60 bis 70% der Fehler in einem Dokument können gefunden werden
- ⇨ Nettonutzen: 20% Ersparnis bei Entwicklung, 30% bei Wartung



Psychologische Probleme bei analytischen Testverfahren:

- ☹ Entwickler sind in aller Regel von der Korrektheit der erzeugten Komponenten überzeugt (ihre Komponenten werden höchstens falsch benutzt)
- ☹ Komponententest wird als lästige Pflicht aufgefasst, die
  - ⇨ Folgearbeiten mit sich bringt (Fehlerbeseitigung)
  - ⇨ Glauben in die eigene Unfehlbarkeit erschüttert
- ☹ Entwickler will eigene Fehler (unbewusst) nicht finden und kann sie auch oft nicht finden (da ggf. sein Testcode von denselben falschen Annahmen ausgeht)
- ☹ Fehlersuche durch getrennte Testabteilung ist noch ärgerlicher (die sind zu doof zum Entwickeln und weisen mir permanent meine Fehlbarkeit nach)
- ☺ Programminspektion und seine Varianten sind u.a. ein Versuch, diese psychologischen Probleme in den Griff zu bekommen
- ☺ Rolle des Moderators ist von entscheidender Bedeutung für konstruktiven Verlauf von Inspektionen



Vorgehensweise bei der Programminspektion:

- **Inspektionsteam** besteht aus Moderator, Autor (passiv), Gutachter(n), Protokoll-führer und ggf. Vorleser (nicht dabei sind Vorgesetzte des Autors)
- **Gutachter** sind in aller Regel selbst (in anderen Projekten) Entwickler
- Inspektion **überprüft**, ob:
  - ⇨ Dokument Spezifikation erfüllt (Implementierung konsistent zu Modell)
  - ⇨ für Dokumenterstellung vorgeschriebene Standards eingehalten wurden
- Inspektion hat **nicht zum Ziel**:
  - ⇨ zu untersuchen, wie entdeckte Fehler behoben werden können
  - ⇨ Beurteilung der Fähigkeiten des Autors
  - ⇨ [lange Diskussion, ob ein entdeckter Fehler tatsächlich ein Fehler ist]
- **Inspektionsergebnis**:
  - ⇨ formalisiertes Inspektionsprotokoll mit Fehlerklassifizierung
  - ⇨ Fehlerstatistiken zur Verbesserung des Entwicklungsprozesses



Ablauf einer Inspektion:

- **Auslösung** der Inspektion durch Autor eines Dokumentes (z.B. durch Freigabe)
- **Eingangsprüfung** durch Moderator (bei vielen offensichtlichen Fehlern wird das Dokument sofort zurückgewiesen)
- **Einführungssitzung**, bei der Prüfling den Gutachtern vorgestellt wird
- **Individualuntersuchung** des Prüflings (Ausschnitt) durch Gutachter anhand ausgeteilter Referenzdokumente (mit Spezifikation, Standards, …)
- auf **Inspektionssitzung** werden Prüfergebnisse mitgeteilt und protokolliert sowie Prüfling gemeinsam untersucht
- **Freigabe** des Prüflings durch Moderator (oder Rückgabe zur Überarbeitung)

Bemerkung:

Bei Individualprüfung werden ca. 80% der Fehler gefunden, bei gemeinsamer Sitzung werden ca. 20% der Fehler gefunden.



Review (abgeschwächte Form der Inspektion):

- Prozessverbesserung und Erstellung von Statistiken steht nicht im Vordergrund
- Moderator gibt Prüfling nicht frei, sondern nur Empfehlung an Manager
- kein formaler Inspektionsplan mit wohldefinierten Inspektionsregeln

Walkthrough:

- Autor des Prüflings liest ihn vor (ablauforientiert im Falle von Software)
- Gutachter versuchen beim Vorlesen ohne weitere Vorbereitung Fehler zu finden
- Autor entscheidet selbst über weitere Vorgehensweise
- Zielsetzungen:
  - ⇨ Fehler/Probleme im Prüfling identifizieren
  - ⇨ Ausbildung/Einarbeitung von Mitarbeitern



9.4 Werkzeugunterstützte statische Testverfahren

(Formale) Verifikation von Software:

Es wird durch mathematisches Beweisverfahren gezeigt, dass Implementierung die Eigenschaften ihrer Spezifikation erfüllt oder dass Spezifikation bestimmte Eigenschaften besitzt.

Probleme:

- sehr zeitaufwändig und deshalb nur in kritischen Fällen durchführbar
- automatische Verifikation mit Werkzeugen auf kleine Softwarekomponenten oder Funktionen beschränkt
- Spezifikation bzw. geforderte Eigenschaften müssen in formaler Form vorliegen
- in Kombination mit UML als Modellierungssprache schwierig, da es zu UML (noch) keine umfassende Semantikdefinition gibt



Werkzeuggestützte Fehlersuche mit statischer Datenflussanalyse:

Die Datenflussanalyse erkennt in Grenzen

- ⇨ deklarierte nicht verwendete Variablen, Parameter, …
- ⇨ lesende Zugriffe auf nicht initialisierte Variable
- ⇨ Zuweisungen von Werten an Variablen, die nie gelesen werden

Probleme mit Programmverzweigungen:

```
IF b1    THEN  a :=  …  ELSE b := … END;
IF b2    THEN  x :=  a   ELSE x : = b END;
```

ist nur dann korrekt, wenn Bedingungen b1 und b2 äquivalent sind.

Das lässt sich meist durch statische Programmanalyse nicht sicherstellen:

- ⇨ **zu viele Fehlermeldungen**, wenn vor lesendem Zugriff auf Variable auf jedem Programmpfad Zuweisung gefordert wird
- ⇨ **zu wenige Fehlermeldungen**, wenn vor lesendem Zugriff auf Variable nur auf einem Programmpfad Zuweisung gefordert wird



Auf Kontrollflussgraph basierende Softwaremetriken:

```
PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
   (* Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)

VAR i: INTEGER;

BEGIN

   count := 0; i := 0;

   WHILE s[i] # '.' DO

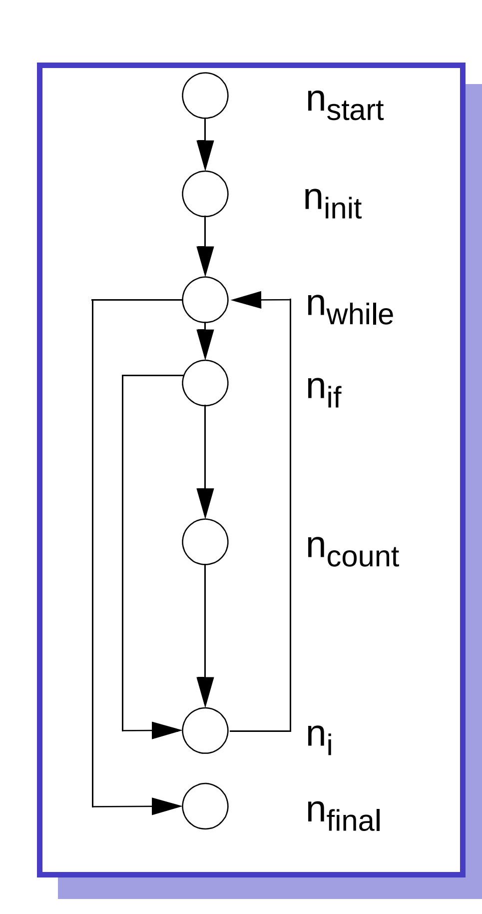
      IF s[i]= 'a' OR s[i]= 'e' OR
         s[i]= 'i' OR s[i]= 'o' OR s[i]= 'u'
      THEN

         count := count + 1;

      END;

      i := i+1;

END countVowels
```



Kontrollflussgraph mit
- 7 **Knoten**
- 8 **Kanten**



Lines of Code (LOC):

LOC(Komponente) = Anzahl der Knoten im Kontrollflussgraphen dazu

Beispiel:

LOC(countVowels) = 7

Idee dieser Maßzahl:

- ⇨ Komponenten mit hoher LOC sind zu komplex (no separation of concerns) und deshalb fehlerträchtig
- ⇨ [Komponenten mit geringer LOC sind zu klein und führen zu unnötigen Schnittstellenproblemen]

Probleme mit dieser Maßzahl:

- ⇨ Kanten = Kontrollflusslogik spielen keine Rolle
- ⇨ wie bewertet man geerbten Code einer Klasse
- ⇨ ...



9.5 Dynamische Testverfahren

Die zu betrachtende Komponente (Operation, Klasse, Paket, Gesamtsystem) wird mit konkreten Eingabewerten ausgeführt und ihr Verhalten wird dabei beobachtet. Im Wesentlichen lassen sich dabei folgende Alternativen unterscheiden:

- **Strukturtest** (white box test): die interne Struktur der Komponente oder ihrer UML-Spezifikation wird zur Testplanung und -Überwachung herangezogen
  - ⇨ **kontrollflussbasiert**: Ablaufverhalten von Operationen wird untersucht
  - ⇨ **datenflussbasiert**: Variablenzugriffe (setzen/lesen) stehen im Vordergrund (werden im Folgenden nicht genauer betrachtet)
  - ⇨ **zustandsbasiert**: Zustände u. Transitionen einer Klasse werden betrachtet
- **Funktionstest** (black box test): die interne Struktur der Komponente wird nicht betrachtet; getestet wird Ein-/Ausgabeverhalten gegen Spezifikation
  - ⇨ z.B. Sequenzdiagramm aus Anwendungsfallbeschreibung oder
  - ⇨ OCL-Ausdruck, der als Nachbedingung Operationsverhalten beschreibt



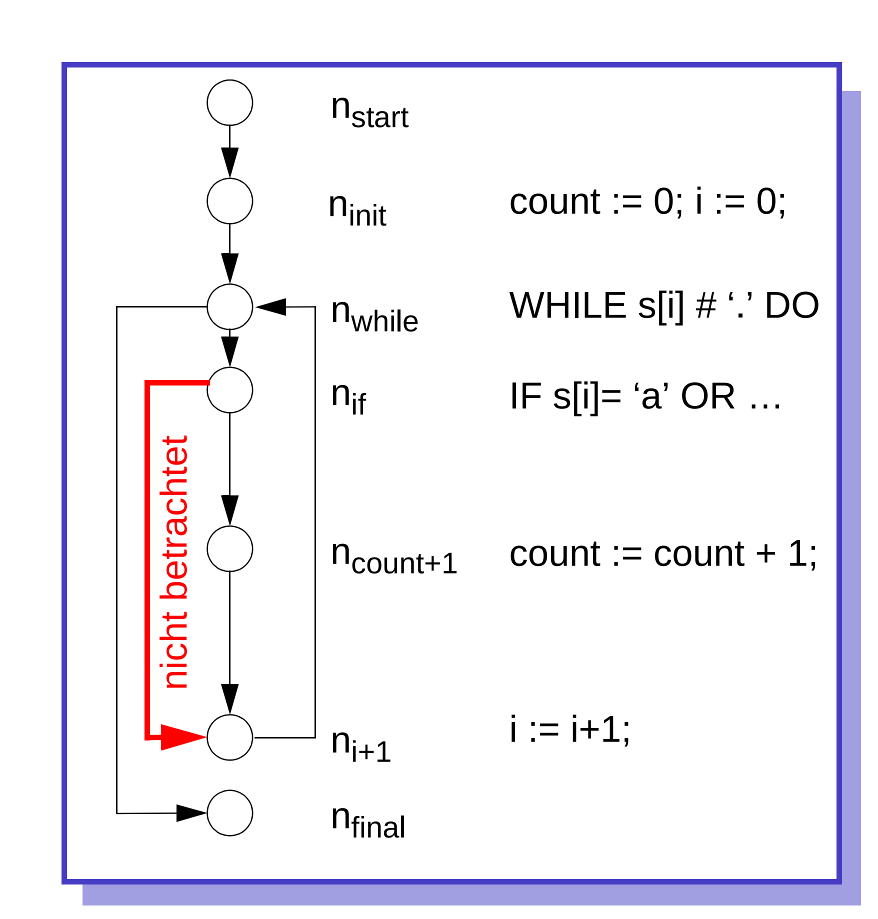
Kontrollflusstest - Anweisungsüberdeckung (C_0-Test):

Jeder Knoten des Kontrollflussgraphen muss mindestens einmal ausgeführt werden.

- Minimalkriterium, da nicht mal alle Kanten des Kontrollflussgraphen traversiert werden
- viele Fehler bleiben unentdeckt

Beispiel:

- bei countVowels genügt als Testfall ein konsonantenfreier Satz wie etwa ‘a.’
- Verschiebung von i := i+1; in if-Anweisung hinein würde nicht als Fehler entdeckt

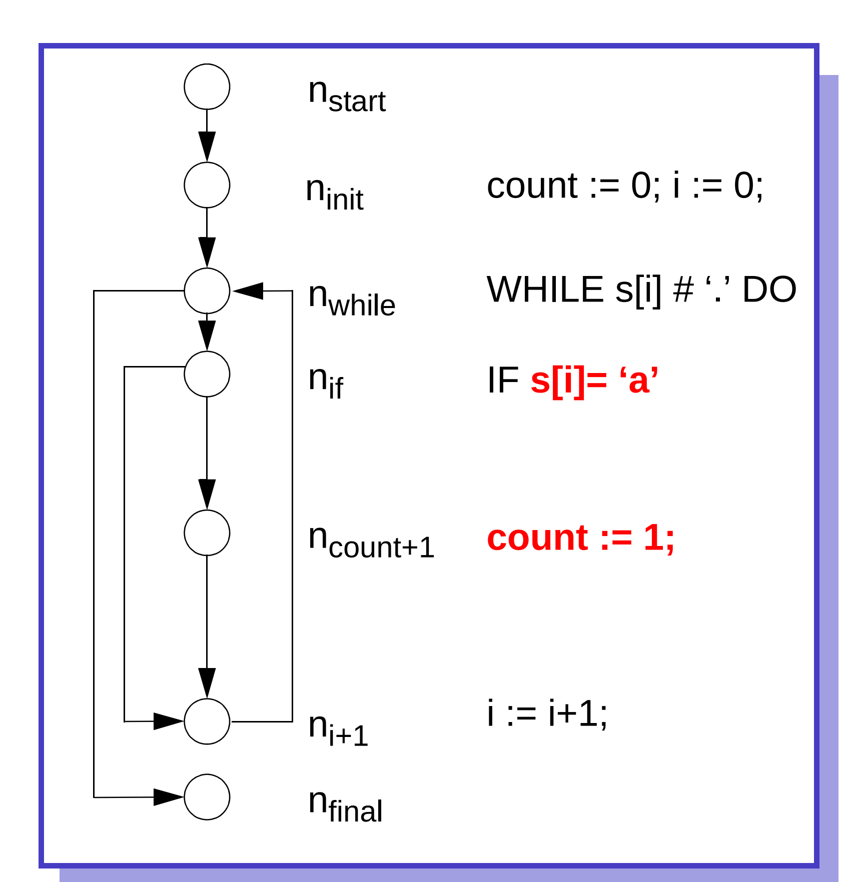




Kontrollflusstest - Zweigüberdeckung (C_1-Test):

Jede Kante des Kontrollflussgraphen muss mindestens einmal ausgeführt werden.

- realistisches Minimalkriterium
- umfasst Anweisungsüberdeckung
- Fehler bei Wiederholung oder anderer Kombination von Zweigen bleiben unentdeckt



Beispiel:

- Testfall 'ab.' für countVowels würde Kriterium bereits erfüllen
- Zuweisung count := 1; würde nicht als fehlerhaft erkannt
- ebenso nicht die verkürzte Bedingung s[i] = 'a'



Funktionstestverfahren (Black-Box-Testen):

Sie testen Implementierung gegen ihre Spezifikation und lassen die interne Programm-struktur unberücksichtigt:

- komplementär zu bislang vorgestellten „White-Box“-Verfahren
- für Abnahmetest ohne Kenntnis des Quellcodes geeignet
- setzt (eigentlich) vollständige und widerspruchsfreie Spezifikation voraus (per se ein Problem bei Verwendung von UML als Modellierungssprache)
- repräsentative Eingabewerte/Szenarien müssen ausgewählt werden (man kann ja im Allgemeinen nicht alle Eingabekombinationen testen)
  - ⇨ Sequenzdiagramme aus Anwendungsfällen beschreiben wichtige Testfälle
- man braucht „Orakel“ für Überprüfung der Korrektheit der Ausgaben (braucht man allerdings bei den „White-Box“-Verfahren auch)
  - ⇨ ausführbare Statecharts als Orakel verwenden



Kriterien für die Auswahl von Eingabewerten:

An der Spezifikation (hier in UML) orientierte **Äquivalenzklassenbildung**, so dass für alle Werte einer Äquivalenzklasse sich das Softwareprodukt „gleich“ verhält:

- Unterteilung in gültige und ungültige Eingabewerte
- gesonderte Betrachtung von Grenzwerten (z.B. Anfang und Ende von Intervallen)
- Auswahl von Repräsentanten so, dass jede Variante jedes Sequenz- und Kollaborationsdiagramms (Aktivitätsdiagramms) einmal durchlaufen wird
- Auswahl von Repräsentanten so, dass jede Transition jedes Statecharts (des Produktautomaten bei and-Zuständen) mindestens einmal schaltet

Achtung:

Auswahl von „Eingabewerten“ schwierig für Objekte als Operationsparameter, z.B. Äquivalenzklassenbildung mit:



Auswahl von Testeingaben für countVowels:

PROCEDURE countVowels(s: Sentence; VAR count: INTEGER);
(Counts how many vowels occur in a sentence. Sentence must be terminated by a dot. *)*

-



9.6 Der Testprozess als Bestandteil des Wasserfallmodells

zur Erinnerung

Machbarkeits-
studie

Anforderungs-
analyse

System-
entwurf

Codieren und
Modultest

Integrations-
& Systemtest

Auslieferung &
Akzeptanztest

Wartung

Entwicklungszeit



Der Testprozess isoliert betrachtet:

Test	Gegenstand	Durchführung
Komponententest	**Prozedur Klasse**	durch **Entwickler**, Testabteilung
Subsystemtest	**Modul Paket**	durch **Entwickler, Testabteilung**
Integrationstest	**Gesamt-system**	durch Entwickler, **Testabteilung**
Akzeptanztest		durch **Auftraggeber**

Hier nur Testen von Code und nicht Testen von (UML-)Modellen betrachtet!



Testen, testen, … - wann ist Schluss damit?

Zitat von Fowler [FS98]:

“The older I get, the more aggressive I get about testing. I like Kent Beck’s rule of thumb that a developer should write at least as much test code as production code. Testing should be a continuous process. No code should be written until you know how to test it. Once you have written it, write the tests for it. Until the test works, you cannot claim to have finished writing the code.”

Wann hört man auf mit Testen:

- ⇨ eines der definierten Testüberdeckungskriterien ist erfüllt
- ⇨ spezifizierte Zuverlässigkeit ist garantiert (siehe nächste Folie)
- ⇨



Spezifikation der Zuverlässigkeit eines Softwaresystems:

1. **Identifikation** aller möglichen Fehlfunktionen eines Softwaresystems (soweit möglich) aus „Black-Box-Sicht“

2. **Klassifikation** der gefundenen Fehlfunktionen, z.B. unter Verwendung der folgenden Attribute:
   - ⇨ vorübergehend, gelegentlich: Fehler treten nur bei gewissen Eingaben auf (permanent, ständig: Fehler treten bei allen Eingaben auf)
   - ⇨ (nicht) wiederherstellbar: ohne (nur mit) Eingreifen des Benutzers kann System nach Fehler wieder in funktionsfähigen Zustand zurückkehren
   - ⇨ (nicht) beschädigend: Fehler beinträchtigt (weder) Systemstatus oder (noch) Daten

3. Für jede Fehlerklasse wird mit einer der eingeführten **Zuverlässigkeitsmetriken** die erforderliche Zuverlässigkeit festgelegt.



Beispiel für eine Zuverlässigkeitsspezifikation:

Fehlerklasse	Beispiel	Zuverlässigkeitsmetrik
permanent	MVRS funktioniert nie am 29. Februar eines Schaltjahres	ROFOC: 1 Vorfall/1000 Tage
gelegentlich, nicht beschädigend	Fahrzeugreservierungsfunktion wird abgebrochen, wenn nicht innerhalb von 5 Minuten alle Eingabedaten vorliegen	POFOD: 1 Vorfall/500 Aufrufe
gelegentlich, beschädigend	Ausmusterung eines Fahrzeugs mit vorliegenden Reservierungen löscht diese Reservierungen	nicht quantifizierbar, sollte nie auftreten

Anmerkung:

Für das Minibeispiel „countVowels“ macht eine Zuverlässigkeitsspezifikation keinen Sinn, deshalb hier Rückgriff auf unser „MotorVehicleReservationSystem“-Beispiel.



Messung der Zuverlässigkeit eines Softwaresystems:

1. Es wird ein **Betriebsprofil** erstellt, das Systemeingaben (Anforderungsfälle) in Klassen unterteilt und Häufigkeiten für diese Klassen festlegt.
2. Ein „hinreichend großer“ **Satz von Testfällen** wird festgelegt, der Betriebsprofil widerspiegelt.
3. Das System wird mit den Testfällen getestet und die **Anzahl der Fehlfunktionen** (die Länge der Ausfallzeiten) wird gemessen.
4. Nachdem eine **statistisch signifikante** Anzahl von Fehlern beobachtet wurde, kann die Zuverlässigkeit der Software berechnet werden.

Probleme bei dieser Vorgehensweise:

- ☹ Erstellung eines Betriebsprofils im allgemeinen sehr schwierig
- ☹ Erstellung einer hinreichenden Menge von Testfällen oft sehr aufwändig
- ☹ „Statistische Signifikanz“ schwer definierbar



9.7 Weitere Literatur

[Ba98] H. Balzert: *Lehrbuch der Softwaretechnik (Band 2): Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum Akademischer Verlag (1998)

Hier findet man (fast) alles Wissenswerte zum Thema Qualitätssicherung und Testen von Software.

[Bi99] R.V. Binder: *Testing Object-Oriented Systems: Models, Pattern, and Tools*, Addison-Wesley (1999)

Bislang umfassendste Monographie zu diesem Themengebiet. Insbesondere zu Testen von Klassen, Berücksichtigung von Automaten sowie zur Testautomatisierung findet ausführliche Informationen. Naturgemäß noch eher knapp gehalten sind die Informationen zum Testen auf Basis von UML-Modellen.

[FS98] M. Fowler, K. Scott: *UML konzentriert: Die neue Standard-Objektmodellierungssprache anwenden*, Addison Wesley (1998), 188 Seiten

Mein Favorit unter den vielen neuen Büchern, die die objektorientierte Softwareentwicklung mit UML zum Thema haben. Schön kompakt, leicht lesbar und mit vielen wertvollen Literaturhinweisen.

[He96] B. Henderson-Sellers: *Object-Oriented Metrics - Measures of Complexity*, Prentice Hall (1996)

Eines der ersten Bücher, das sich mit der Definition von Softwaremetriken für objektorientierte Programme (Modelle) befaßt.

[So01] I. Sommerville: *Software Engineering*, Addison-Wesley - Pearson Studium, 6. Auflage (2001), 711 Seiten

Wieder aktuelles in Deutsch übersetztes Lehrbuch, das sehr umfassend alle wichtigen Themen der Software-Technik knapp behandelt. Empfehlenswert!



10. Management der Software-Entwicklung

Themen dieses Kapitels:

- Fehlende Phasen des Wasserfallmodells
- bessere/modernere Prozessmodelle
- Verbesserung/Qualität von Softwareprozessmodellen
- Kostenschätzung (Zeitplanung) für Entwicklungsprozesse
- Projektmanagement(-werkzeuge)

Achtung:

Viele im folgenden vorgestellten Überlegungen sind nicht ausschließlich für **Software**-Entwicklungsprozesse geeignet, sondern werden ganz allgemein für die Steuerung komplexer technischer Entwicklungsprozesse eingesetzt.



Aufgaben des Managements:

- **Planungsaktivitäten**: Ziele definieren, Vorgehensweisen auswählen, Termine fest-legen, Budgets vorbereiten, …
  - ⇨ Vorgehensmodelle, Kostenschätzung, Projektpläne
- **Organisationsaktivitäten**: Strukturieren von Aufgaben, Festlegung organisatori-scher Strukturen, Definition von Qualifikationsprofilen für Positionen, …
  - ⇨ Rollenmodelle, Team-Modelle, Projektpläne
- **Personalaktivitäten**: Positionen besetzen, Mitarbeiter beurteilen, weiterbilden, …
  - ⇨ nicht Thema dieser Vorlesung
- **Leitungsaktivitäten**: Mitarbeiter führen, motivieren, koordinieren, …
  - ⇨ nicht Thema dieser Vorlesung
- **Kontrollaktivitäten**: Prozess- und Produktstandards entwickeln, Berichts- und Kontrollwesen etablieren, Prozesse und Produkte vermessen, Korrekturen, …
  - ⇨ Qualititätsmanagement, insbesondere für Software-Entwicklungsprozesse



Ziele des Managements:

Hauptziel des Projektmanagements ist die **Erhöhung der Produktivität**!

Allgemeine Definition von Produktivität:

Produktivität = Produktwert / Aufwand (oder: Leistung / Aufwand)

Für Software-Entwicklung oft verwendete Definition:

Produktivität = Größe der Software / geleistete Mitarbeitertage

Probleme mit dieser Definition:

- ⇨ Maß für Größe der Software
- ⇨ Berücksichtigung der Produktqualität
- ⇨ Aufwand = Mitarbeitertage ?
- ⇨ Nutzen (Return Of Investment) = Größe der Software ?



Einflussfaktoren für Produktivität [ACF97]:

Angabe der Form “+ 1 : X” steht für Produktivitätssteigerung um maximal Faktor X
Angabe der Form “- 1 : Y” steht für Produktivitätsminderung um maximal Faktor Y.

- ⇨ Werkzeug-Einsatz:
- ⇨ Geeignete Methoden:
- ⇨ Produktkomplexität:
- ⇨ Hohe Zuverlässigkeitsanforderung:
- ⇨ Firmenkultur (corporate identity):
- ⇨ Arbeitsumgebung (eigenes Büro, abschaltbares Telefon, …): +
- ⇨ Begabung der Mitarbeiter:
- ⇨ Erfahrung im Anwendungsgebiet:
- ⇨ Bezahlung, Berufserfahrung:



Produktivitäts- und Qualitätsverlauf bei Einsatz neuer Werkzeuge [Jo92]:

Qualität = Anzahl entdeckter Fehler / Quellcodezeilen ?

Produktivität = Produktwert / Kosten
oder = Quellcodezeilen / Mitarbeitermonate

Nach anderen Untersuchungen liegt die maximale Produktivitätssteigerung bei *optimalem* Einsatz von Werkzeugen (CASE-Tools) bei 50 bis 60 %.



10.1 Zurück zum Wasserfallmodell

Machbar-keitsstudie → Anforderungs-analyse → System-entwurf → Codieren und Modultest → Integrations- & Systemtest → **Auslieferung & Installation** → **Wartung**

- Machbarkeitsstudie: siehe Kapitel 3 und Kapitel 10
- Anforderungsanalyse: siehe Kapitel 5 und Kapitel 7
- Systementwurf, Codieren und Modultest: siehe Kapitel 7 und Kapitel 8
- Integrations- & Systemtest: siehe Kapitel 9
- Auslieferung & Installation, Wartung: siehe Kapitel 10

bislang nicht (vollständig) besprochen → Auslieferung & Installation, Wartung



Die Abnahmephase (Auslieferung):

Diese Phase beendet die Entwicklung (einer Generation) eines Softwareproduktes durch die vertragsgemäße Übergabe an den Auftraggeber. Es werden folgende Aktivitäten durchgeführt:

- **Übergabe des Produkts**:
  - ⇨ Produktion für den anonymen Markt: Lizenz wird mit Binärcode und Dokumentation übergeben, ggf. Wartungsvertrag für Bezug künftiger Softwareversionen und Benutzung von „Hotline"
  - ⇨ Produktion von Individualsoftware (mit Wartung): Binärcode der Software wird mit Dokumentation übergeben, Quellcode und Wartungsverpflichtung bleibt beim Auftragnehmer
  - ⇨ Produktion von Individualsoftware (ohne Wartung): alle Entwicklungsdokumente (Analysedokumente, Quellcode, etc.) werden zusammen mit dem Binärcode und der Benutzerdokumentation übergeben



Die Abnahmephase (Auslieferung) - Fortsetzung:

- **Abnahmetest**:

  Auftraggeber überprüft die (vertraglich vereinbarten) Abnahmekriterien

  - ⇨ Wartung verbleibt bei Auftragnehmer: Benutzbarkeit, Effizienz, Zuverlässigkeit, Korrektheit, … werden getestet
  - ⇨ Wartung geht auf Auftraggeber über: zusätzlich spielen Wartbarkeit, Portierbarkeit, Testbarkeit, Erweiterbarkeit, … eine wichtige Rolle

- **Abnahmeprotokoll**:

  In ihm werden die durchgeführten Abnahmetests und ihre Ergebnisse festgehalten.

- **Abnahme**:

  Schriftliche Erklärung der Annahme des Softwareprodukts (im juristischen Sinne).



Einführungsphase (Installation):

In der Einführungsphase werden (nach der Abnahme des Softwareprodukts) folgende Aktivitäten durchgeführt:

- **Installation**: Einrichtung des Produkts beim Kunden (auf Zielumgebung) zum Zwecke der Inbetriebnahme (Software für anonymen Markt: Pilotinstallationen = Betatest)
- **Schulung**: nach der Installation werden Benutzer und Betriebspersonal in die Bedienung des Produkts eingewiesen (Software für anonymen Markt: separater Vertrieb entsprechender Kurse)
- **Inbetriebnahme**: Übergang zwischen Installation und vollem Betrieb, dabei ggf. Ablösung eines Altsystems:
  - ⇨ direkte Umstellung (riskant): Altsystem wird stillgelegt, alle Daten übertragen und dann Vollbetrieb des neuen Systems
  - ⇨ Parallelllauf (aufwändig): Altsystem und neues System laufen parallel, ihre Daten müssen dauernd konsistent gehalten werden



Wartungs- und Pflegephase:

Mit der Einführung einer Software in den produktiven Betrieb beginnt die Wartung und Pflege von Software:

- ⇨ unter **Wartung** versteht man das Beseitigen von Fehlern
- ⇨ unter **Pflege** versteht man die Anpassung und Erweiterung von Software

Es gibt aber auch die Unterteilung der Wartung in:

- ⇨ ca. 20% **korrigierende** Aktivitäten (corrective maintenance): Behebung von Fehlern und Inkonsistenzen zwischen Implementierung u. Lastenheft
- ⇨ ca. 25% **anpassende** Aktivitäten (adaptive maintenance): Anpassung an neue Rahmenbedingungen (Portierung auf neue Hardware, …)
- ⇨ ca. 55% **perfektionierende** Maßnahmen (perfective maintenance): Verbesserung der Effizienz und Wartbarkeit der Software

Untersuchungen besagen, dass Aufwand für Wartung und Pflege **doppelt bis viermal** so hoch wie Aufwand für Entwicklung ist.



Typische Probleme in der Wartungsphase:

- Einsatz wenig erfahrenen Personals (nicht Entwicklungspersonal)
- Fehlerbehebung führt neue Fehler ein
- stetige Verschlechterung der Programmstruktur
- Zusammenhang zwischen Programm und Dokumentation geht verloren
- zur Entwicklung eingesetzte Werkzeuge (CASE-Tools, Compiler, …) sterben aus
- benötigte Hardware steht nicht mehr zur Verfügung
- Resourcenkonflikte zwischen Fehlerbehebung und Anpassung/Erweiterung
- völlig neue Ansprüche an Funktionalität und Benutzeroberfläche

Konsequenz:

⇨ **Rechtzeitige Sanierung oder Neuentwicklung von Softwareprodukt!**

⇨ … und (bessere) Integration der Wartungsphase in Vorgehensmodell



Prozessmodell für die Wartung:



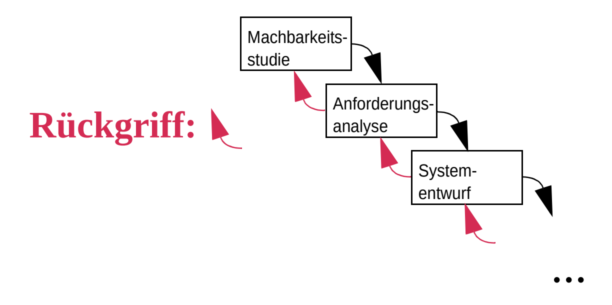
Probleme mit dem Wasserfallmodell insgesamt:

- ☹ Wartung mit ca. 60% des Gesamtaufwandes ist eine Phase
  - ⇨ andere Prozessmodelle mit Wartung als eigener Entwicklungsprozess
- ☹ zu Projektbeginn sind nur ungenaue Kosten- und Ressourcenschätzungen möglich
  - ⇨ Methoden zur Kostenschätzung anhand von Lastenheft (Pflichtenheft)
- ☹ ein Pflichtenheft kann nie den Umgang mit dem fertigen System ersetzen, das erst sehr spät entsteht (Risikomaximierung)
  - ⇨ andere Prozessmodelle mit Erstellung von Prototypen, …
- ☹ Anforderungen werden früh eingefroren, notwendiger Wandel (aufgrund organisatorischer, politischer, technischer, … Änderungen) nicht eingeplant
  - ⇨ andere Prozessmodelle mit evolutionärer Software-Entwicklung
- ☹ strikte Phaseneinteilung ist unrealistisch (Rückgriffe sind notwendig)
  - ⇨ andere Prozessmodelle mit iterativer Vorgehensweise



10.2 Andere Vorgehensmodelle

Die naheliegendste Idee zur Verbesserung des Wasserfallmodells ergibt sich durch die Einführung von **Zyklen** bzw. **Rückgriffen**. Sie erlauben Wiederaufnehmen früherer Phasen, wenn in späteren Phasen Probleme auftreten.



Weitere Vorgehensmodelle:

- das V-Modell (umgeklapptes Wasserfallmodell)
- das evolutionäre Modell (iteriertes Wasserfallmodell)
- Rapid Prototyping (Throw-Away-Prototyping)



Das V-Modell:

SE 1: System-Anforderungs-analyse

SE 2: System-Entwurf

SE 3: SW-HW-Anforderungs-analyse

SE 4: SW-Grobentwurf

SE 5: SW-Feinentwurf

SE 6: SW-Implementierung

SE 7: SW-Integration

SE 8: System-Integration

SE 9: Überleitung in die Nutzung

Überblick zum V-Modell 97 findet man in [OHJ99], neu ist das V-Modell XT



Beschreibung der einzelnen Phasen des V-Modells:

- **Systemanforderungsanalyse**: Gesamtsystem einschließlich aller Nicht-DV-Komponenten wird beschrieben (fachliche Anforderungen und Risikoanalyse)
- **Systementwurf**: System wird in technische Komponenten (Subsysteme) zerlegt, also die Grobarchitektur des Systems definiert
- **Softwareanforderungsanalyse**: technischen Anforderungen an die bereits identifizierten Komponenten werden definiert
- **Softwaregrobentwurf**: Softwarearchitektur wird bis auf Modulebene festgelegt
- **Softwarefeinentwurf**: Details einzelner Module werden festgelegt
- **Softwareimplementierung**: wie beim Wasserfallmodell (inklusive Modultest)
- **Software-/Systemintegration**: schrittweise Integration und Test der verschiedenen Systemanteile
- **Überleitung in die Nutzung**: entspricht Auslieferung bei Wasserfallmodell

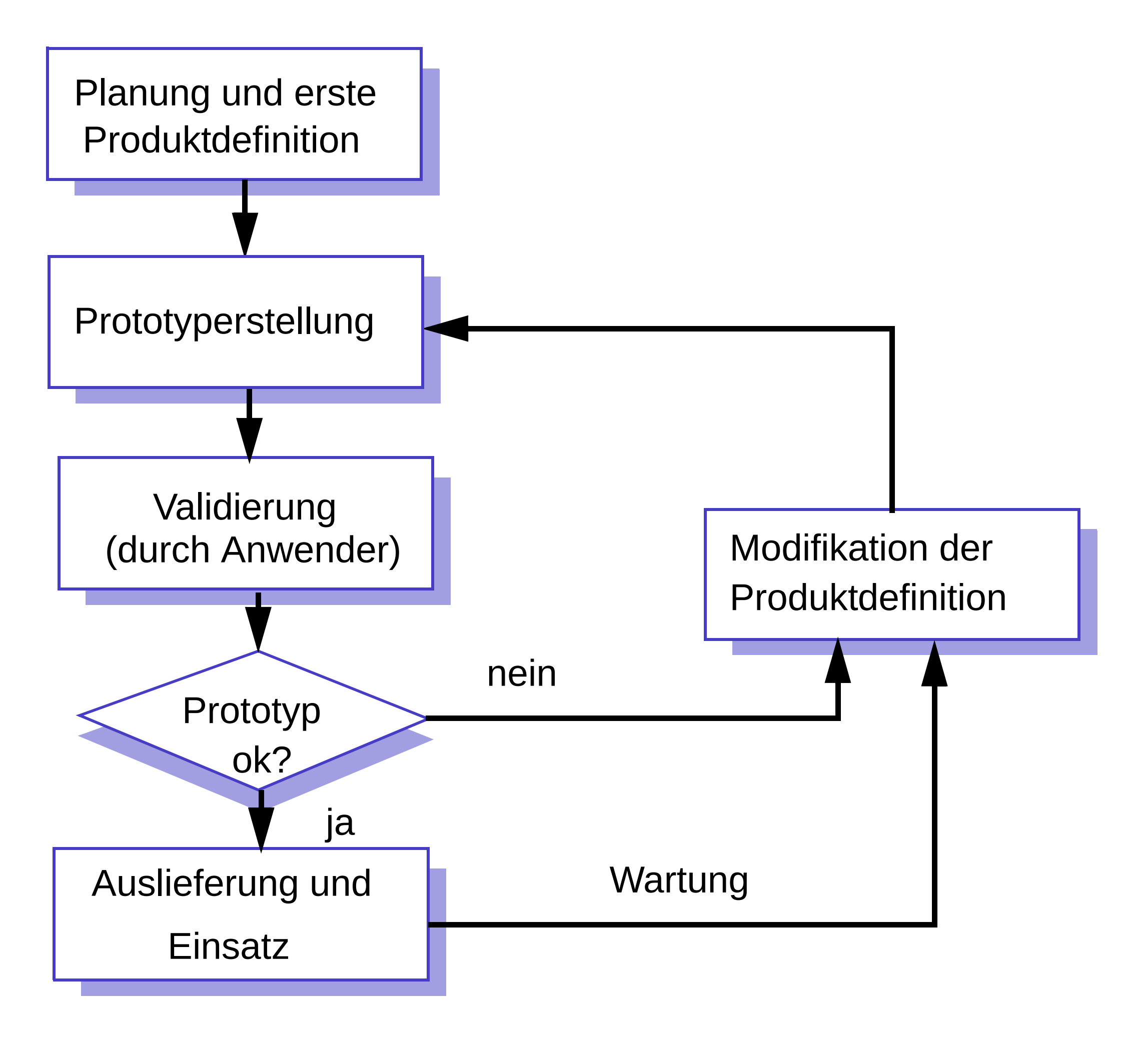


Bewertung des V-Modells:

- ☺ erfasst auch Systemkomponenten die nicht in Software realisiert werden
- ☺ Standarddefinition ausserordentlich umfangreich mit vielen Teilaktivitäten
- ☺ Qualitätssicherung, Konfigurationsmanagement, … wird auch unterstützt
- ☺ korrespondierende Phasen des Wasserfallmodells einander zugeordnet
- ☺ “allumfassende” Wartungsphase entfällt zu recht
- ☹ grundsätzliche Schwächen der strikten Phaseneinteilung des Wasserfallmodells bleiben auf den ersten Blick erhalten
  - ⇨ iterativ-inkrementelle Vorgehensweise wird aber unterstützt, in der Gesamtentwicklung in Produktzyklen unterteilt wird
- ☹ Anwender beklagen (zu) starke Reglementierung und Papierflut
  - ⇨ Variantenbildung wird aber unterstützt, die Anpassungen an Produktum-fang, Firmenkultur, … erlaubt



Evolutionäres Modell (evolutionäres Prototyping):





Bewertung des evolutionären Modells:

- ☺ es ist sehr früh ein (durch Kunden) evaluierbarer Prototyp da
- ☺ Kosten und Leistungsumfang des gesamten Softwaresystems müssen nicht zu Beginn des Projekts vollständig festgelegt werden
- ☺ Projektplanung vereinfacht sich durch überschaubarere Teilprojekte
- ☺ Systemarchitektur muss auf Erweiterbarkeit angelegt sein
- ☹ es ist schwer, Systemarchitektur des ersten Prototypen so zu gestalten, dass sie alle später notwendigen Erweiterungen erlaubt
- ☹ Prozess der Prototyperstellung nicht festgelegt
  - ⇨ Spiralmodell von Berry Böhm integriert Phasen des Wasserfallmodells
- ☹ evolutionäre Entwicklung der Anforderungsdefinition birgt Gefahr in sich, dass bereits realisierte Funktionen hinfällig werden
- ☹ Endresultat sieht ggf. wie Software nach 10 Jahren Wartung aus



Rapid Prototyping (Throw-Away-Prototyping):

Mit Generatoren, ausführbaren Spezifikationssprachen, Skriptsprachen etc. wird

- ☞ Prototyp des Systems (seiner Benutzeroberfläche) realisiert
- ☞ dem Kunden demonstriert
- ☞ und anschließend weggeschmissen

Bewertung:

- ☺ erlaubt schnelle Klärung der Funktionalität und Risikominimierung
- ☺ Vermeidung von Missverständnissen zwischen Entwickler und Auftraggeber
- ☺ früher Test der Benutzerschnittstelle



10.3 Rational Unified Process für UML

Product Life Cycle (Stand 1999)

1st generation | 2nd generation | 3rd generation | …

Manager-sicht

inception phase | elaboration phase | construction phase | transition phase

Entwickler-sicht

iteration 1 | iteration 2 | iteration 3 | iteration 4 | iteration 5 | …

planning | analysis | design | implementation | test

☞ Firma IBM (ehemals Rational) dominiert(e) Entwicklung der Standard-OO-Modellierungssprache UML und des zugehörigen Vorgehensmodells.



Phasen der Lebenszyklusgenerationen:

- **Inception (Vorbereitung)**: Definition des Problembereichs und Projektziels für Produktgeneration mit Anwendungsbereichsanalyse (Domain Analysis) und Machbarkeitsstudie (für erste Generation aufwändiger)
  - ⇨ bei Erfolg weiter zu …
- **Elaboration (Entwurf)**: erste Anforderungsdefinition für Produktgeneration mit grober Softwarearchitektur und Projektplan (ggf. mit Rapid Prototyping)
  - ⇨ bei Erfolg weiter zu …
- **Construction (Konstruktion)**: Entwicklung der neuen Produktgeneration als eine Abfolge von Iterationen mit Detailanalyse, -design, … (wie beim evolutionären Vorgehensmodell)
  - ⇨ bei Erfolg weiter zu …
- **Transition (Einführung)**: Auslieferung des Systems an den Anwender (inklusive Marketing, Support, Dokumentation, Schulung, …)



Eigenschaften des Rational Unified Prozesses (RUP):

- **modellbasiert**: für die einzelnen Schritte des Prozesses ist festgelegt, welche Modelle (Dokumente) des Produkts zu erzeugen sind
- **prozessorientiert**: die Arbeit ist in eine genau definierte Abfolge von Aktivitäten unterteilt, die von anderen Teams in anderen Projekten wiederholt werden können.
- **iterativ und inkrementell**: die Arbeit ist in eine Vielzahl von Iterationen unter-teilt, das Produkt wird inkrementell entwickelt.
- **risikobewusst**: Aktivitäten mit hohem Risiko werden identifiziert und in frühen Iterationen in Angriff genommen.
- **zyklisch**: die Produktentwicklung erfolgt in Zyklen (Generationen). Jeder Zyklus liefert eine neue als kommerzielles Produkt ausgelieferte Systemgeneration.
- **ergebnisorientiert**: jede Phase (Iteration) ist mit der Ablieferung eines definierten Ergebnisses meist zu einem konkreten Zeitpunkt (Meilenstein) verbunden



Faustregeln für die Ausgestaltung eines Entwicklungsprozesses:

- die Entwicklung einer **Produktgeneration** dauert höchstens 18 Monate
- eine **Vorbereitungsphase** dauert 3-6 Wochen und besteht aus einer Iteration
- eine **Entwurfsphase** dauert 1-3 Monate und besteht aus bis zu 2 Iterationen
- eine **Konstruktionsphase** dauert 1-9 Monate und besteht aus bis zu 7 Iterationen
- eine **Einführungsphase** dauert 4-8 Wochen und besteht aus einer Iteration
- jede **Iteration** dauert 4-8 Wochen (ggf. exklusive Vorbereitungs- und Nachbereitungszeiten, die mit anderen Iterationen überlappen dürfen)
- das gewünschte **Ergebnis** (Software-Release) einer Iteration ist spätestens bei ihrem Beginn festgelegt (oft Abhängigkeit von Ergebnissen vorheriger Iterationen)
- die **geplante Zeit** für eine Iteration wird nie (höchstens um 50%) überschritten
- innerhalb der Konstruktionsphase wird mindestens im wöchentlichen Abstand ein **internes Software-Release** erstellt
- mindestens 40% **Reserve** an Projektlaufzeit für unbekannte Anforderungen, …



Rollenbasierte Softwareentwicklung und Arbeitsbereiche - Wiederholung:

- eine **Rolle (WorkerKind)** beschreibt eine Menge von eng zusammengehörigen Aufgaben und Verantwortlichkeiten (oft auch notwendige Qualifikationen)
- **Personen** (im Extremfall auch Werkzeuge) nehmen Rollen ein
- ein **Arbeitsbereich (ActivityKind)** umfasst eine Menge eng zusammengehöriger Aufgaben und Verantwortlichkeiten
- zu einem Arbeitsbereich gehört eine Menge voneinander abhängiger **Artefakte (ArtifactKind)**, die als Eingabe benötigt oder als Ausgabe produziert werden
- **Aufgaben** zur Bearbeitung von Artifakten werden von Personen in bestimmten Rollen (gebunden an einen Arbeitsbereich) durchgeführt



Anmerkungen zu den Arbeitsbereichen (Workflows) des RUP:

- **Business Modeling** befasst sich damit, das Umfeld des zu erstellenden Software-systems zu erfassen (Geschäftsvorfälle, Abläufe, …)
- **Requirements (Capture)** befasst sich damit die Anforderungen an ein Software-system noch sehr informell zu erfassen
- **Analysis/Design** präzisiert mit grafischen Sprachen (Klassendiagramme etc.) die Anforderungen und liefert Systemarchitektur
- **Implementation/Test** entspricht den Aktivitäten in den Phasen „Codierung bis Integrationstest" des Wasserfallmodells
- **Deployment** entspricht „Auslieferung und Installation" des Wasserfallmodells
- **Configuration Management** befasst sich mit der Verwaltung von Softwareversio-nen und -varianten (siehe Vorlesung „Software Engineering II")
- **Project Management** mit der Steuerung des Entwicklungsprozesses selbst
- **Environment** bezeichnet die Aktivitäten zur Bereitstellung benötigter Ressourcen (Rechner, Werkzeuge, …)



Bewertung des (Rational) Unified Prozesses:

- ☺ es gibt Standardisierungsvorschlag für OMG und kommerzielles Produkt (WWW-basiert: **http://www-306.ibm.com/software/awdtools/rup/**)
- ☺ Manager hat die grobe „Inception-Elaboration-Construction-Transition”-Sicht
- ☺ Entwickler hat zusätzlich die feinere arbeitsbereichsorientierte Sicht
- ☺ Wartung ist eine Abfolge zu entwickelnder Produktgenerationen
- ☺ es wird endgültig die Illusion aufgegeben, dass Analyse, Design, … zeitlich begrenzte strikt aufeinander folgende Phasen sind
- ☹ komplexes noch in Veränderung befindliches Vorgehensmodell für UML
- ☹ noch nicht mit Behördenstandards (V-Modell, …) richtig integriert
- ☹ Qualitätssicherung ist kein eigener Aktivitätsbereich



10.4 Leichtgewichtige Prozessmodelle

Herkömmlichen Standards für Vorgehensmodelle zur Softwareentwicklung (V-Modell, RUP) wird vorgeworfen, dass

- sie sehr starr sind
- Unmengen an Papier produzieren
- und nutzlos Arbeitskräfte binden

Deshalb werden seit einiger Zeit sogenannte „leichtgewichtige“ Prozessmodelle (**light-weight processes**) propagiert, die sinnlosen bürokratischen Overhead vermeiden wollen:

- **Extreme Programming (XP)**: radikale Abkehr von bisherigen Vorgehens-modellen und Ersatz durch „best practices“ der Programmierung [Be99]
- **Agile Prozessmodelle**: Versuche, herkömmliche Prozessmodelle auf das unbedingt notwendige zurückzuschneiden und situationsbedingt flexibel und schnell (agil) voranzuschreiten



Grundideen von XP:

- Verzicht auf Phasen der Softwareentwicklung, Arbeitsbereiche, …
- keine eigenständigen Analyse- oder Designaktivitäten vor der Codierung
- keine Erstellung getrennter Dokumentationsdokumente, Modelle, …
- der (gut kommentierte) Code ist seine eigene Dokumentation
- aber großes Gewicht auf sorgfältige Testplanung
- Tests werden immer zusammen mit (oder gar vor) Code geschrieben
- nach jeder Änderung werden alle Tests durchlaufen (Regressionstests)
- unbedingtes Einhalten der 40-Stunden-Woche
- Auftraggeber soll permanent neue Anforderungen aufstellen und priorisieren
- sehr kurze Iterationszyklen, häufige Releases
- „Pair Programming“ Teams
- Subsets obiger Ideen dürfen nicht XP genannt werden



Randbedingungen:

- maximal 5 bis 10 Programmierer an einem Projekt beteiligt
  - ⇨ Beschränkung auf relativ kleine Projekte
- Kommunikation zwischen Programmierern und mit dem Kunden sehr intensiv
  - ⇨ keine räumlich verteilten Teams
- Kunde verzichtet auf separate Dokumentation der Software
  - ⇨ zugunsten einer durch ausführliches Testen unterstützten Rapid-Prototyping-Vorgehensweise
- Programmierer versuchen nicht, bei der Realisierung des aktuellen Releases bereits künftige Releases mit zu berücksichtigen
- Programmierer sind aber zum ständigen Umbau (Redesign) der bereits erstellten Software bereit
  - ⇨ Prinzip des Refactorings von Software

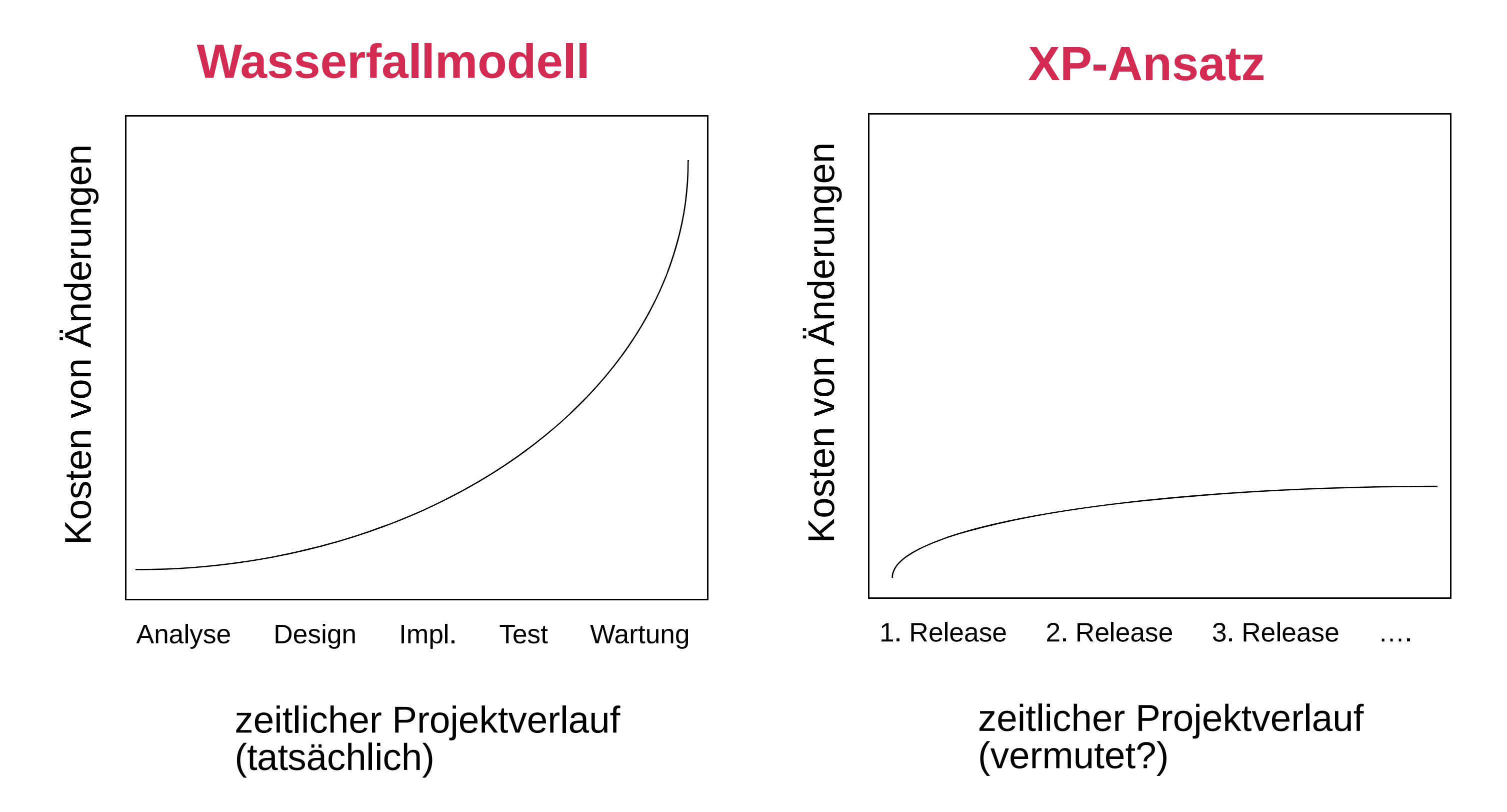


Bewertung von XP:

- ☺ systematisches Hacking: Kompromiss zwischen Wirklichkeit der Programmentwicklung in vielen Fällen und sehr aufwändigen Vorgehensmodellen
- ☺ Betonung der menschlichen Komponente: 40-Stunden-Woche, intensive Kommunikation, Pair Programming, Empfehlungen für Arbeitsumgebung
- ☺ systematisches evolutionäres Rapid Prototyping: durch Testen und Refactoring (Software-Sanierung in kleinen Schritten) versucht man Nachteile zu vermeiden
- ☹ Endprodukt besitzt (ausser Kommentaren im Quellcode und Testfälle) keine Dokumentation
- ☹ Entwicklung des benötigten Gesamtsystems wird durch schnelle Produktion vieler kleiner Releases nicht unbedingt sehr zielgerichtet sein
- ☹ Grundannahmen (z.B. leichte Änderbarkeit des so erstellten Codes über gesamte Projektlaufzeit hinweg) nicht hinreichend empirisch belegt
- ☹ nur für kleine Projekte in nicht sicherheitskritschen Anwendungsbereichen



Lineare Kostenkurve:



Bei „klassischen" Projekten steigen die Kosten für geänderte Anforderungen mit Abschluss jeder Phase deutlich an, bei XP verspricht man sich eine konstante Kosten-kurve für die Durchführung gewünschter Änderungen (da nur Code zu ändern ist).



10.5 Verbesserung der Prozessqualität

Ausgangspunkt der hier vorgestellten Ansätze sind folgende Überlegungen:

- ⇨ Softwareentwicklungsprozesse sind selbst Produkte, deren Qualität überwacht und verbessert werden muss
- ⇨ bei der Softwareentwicklung sind bestimmte Standards einzuhalten (Entwicklungsprozess muss dokumentiert und nachvollziehbar sein)
- ⇨ es bedarf kontinuierlicher Anstregungen, um die Schwächen von Entwicklungsprozessen zu identifizieren und zu eliminieren

Hier vorgestellte Ansätze:

- ⇨ **ISO 9000 Normenwerk** (Int. Standard für die Softwareindustrie)
- ⇨ **Capability Maturity Model** (CMM) des Software Engineering Institutes (SEI) an der Carnegie Mellon University



Qualitätssicherung mit der ISO 9000:

Das **ISO 9000 Normenwerk** (http://www.iso-9000.co.uk/) legt für das Auftraggeber-Lieferantenverhältnis einen allgemeinen organisatorischen Rahmen zur Qualitätssicherung fest.

Das **ISO 9000 Zertifikat** bestätigt, dass die Verfahren eines Unternehmens der ISO 9000 Norm entsprechen.

Wichtige Teile:

- **ISO 9000-1**: allgemeine Einführung und Überblick
- **ISO 9000-3**: Anwendung von ISO 9001 auf Softwareproduktion
- **ISO 9001**: Modelle der Qualitätssicherung in Design/Entwicklung, Produktion, Montage und Kundendienst
- **ISO 9004**: Aufbau und Verbesserung eines Qualitätsmanagementsystems



Von ISO 9000-3 vorgeschriebene Dokumente:

- **Vertrag Auftraggeber - Lieferant**: Tätigkeiten des Auftraggebers, Behandlung von Anforderungsänderungen, Annahmekriterien (Abnahmetest), …
- **Spezifikation**: funktionale Anforderungen, Ausfallsicherheit, Schnittstellen, … des Softwareprodukts
- **Entwicklungsplan**: Zielfestlegung, Projektmittel, Entwicklungsphasen, Management, eingesetzte Methoden und Werkzeuge, …
- **Qualitätssicherungsplan**: Qualitätsziele (messbare Größen), Kriterien für Ergebnisse v. Entwicklungsphasen, Planung von Tests, Verifikation, Inspektionen
- **Testplan**: Teststrategie (für Integrationstest), Testfälle, Testwerkzeuge, Kriterien für Testvollständigkeit/Testende
- **Wartungsplan**: Umfang, Art der Unterstützung, …
- **Konfigurationsmanagement**: Plan für Verwaltung von Softwareversionen und Softwarevarianten



Von ISO 9000-3 vorgeschriebene Tätigkeiten:

- **Konfigurationsmanagement** für Identifikation und Rückverfolgung von Änderungen, Verwaltung parallel existierender Varianten
- **Dokumentmanagement** für geordnete Ablage und Verwaltung aller bei der Softwareentwicklung erzeugten Dokumente
- **Qualitätsaufzeichnungen** (Fehleranzahl oder Metriken) für Verbesserungen am Produkt und Prozess
- **Festlegung** von Regeln, Praktiken und Übereinkommen für ein Qualitäts-sicherungssystem
- **Schulung** aller Mitarbeiter sowie Verfahren zur Ermittlung des Schulungsbedarfes

Zertifizierung:

Die Einhaltung der Richtlinien der Norm wird von unabhängigen Zertifizierungs-stellen im jährlichen Rhythmus überprüft.



Bewertung von ISO 9000:

- ☺ lenkt die Aufmerksamkeit des Managements auf Qualitätssicherung
- ☺ ist ein gutes Marketing-Instrument
- ☺ reduziert das Produkthaftungsrisiko (Nachvollziehbarkeit von Entscheidungen)
- ☹ Nachvollziehbarkeit und Dokumentation von Prozessen reicht aus
- ☹ keine Aussage über Qualität von Prozessen und Produkten
- ☹ (für kleine Firmen) nicht bezahlbarer bürokratischer Aufwand
- ☹ Qualifikation der Zertifizierungsstellen umstritten
- ☹ oft große Abweichungen zwischen zertifiziertem Prozess und realem Prozess



Das Capability Maturity Model (CMM):

Referenzmodell zur Beurteilung von Softwarelieferanten, vom Software Engineering Institute entwickelt (http://www.sei.cmu.edu/cmm/cmms.html).

- ⇨ Softwareentwicklungsprozesse werden in **5 Reifegrade** unterteilt
- ⇨ Reifegrad (**maturity**) entspricht Qualitätsstufe der Softwareentwicklung
- ⇨ höhere Stufe beinhaltet Anforderungen der tieferen Stufen

Stufe 1 - chaotischer initialer Prozess (ihr Stand vor dieser Vorlesung):

- ❑ Prozess-Charakteristika:
  - ⇨ unvorhersehbare Entwicklungskosten, -zeit und -qualität
  - ⇨ kein Projektmanagement, nur „Künstler“ am Werke
- ❑ notwendige Aktionen:
  - ⇨ Planung mit Kosten- und Zeitschätzung einführen
  - ⇨ Änderungs- und Qualitätssicherungsmanagement



Stufe 2 - wiederholbarer intuitiver Prozess (Stand nach dieser Vorlesung?):

- Prozess-Charakteristika:
  - ⇨ Kosten und Qualität schwanken, gute Terminkontrolle
  - ⇨ Know-How einzelner Personen entscheidend
- notwendige Aktionen:
  - ⇨ Prozessstandards entwickeln
  - ⇨ Methoden (für Analyse, Entwurf, Testen, …) einführen

Stufe 3 - definierter qualitativer Prozess (Stand der US-Industrie 1989?):

- Prozess-Charakteristika:
  - ⇨ zuverlässige Kosten- und Terminkontrolle, schwankende Qualität
  - ⇨ institutionalisierter Prozess, unabhängig von Individuen
- notwendige Aktionen:
  - ⇨ Prozesse vermessen und analysieren
  - ⇨ quantitative Qualitätssicherung



Stufe 4 - gesteuerter/geleiteter quantitativer Prozess:

- Prozess-Charakteristika:
  - ⇨ gute statistische Kontrolle über Produktqualität
  - ⇨ Prozesse durch Metriken gesteuert
- notwendige Aktionen:
  - ⇨ instrumentierte Prozessumgebung (mit Überwachung)
  - ⇨ ökonomisch gerechtfertigte Investitionen in neue Technologien

Stufe 5 - optimierender rückgekoppelter Prozess:

- Prozess-Charakteristika:
  - ⇨ quantitative Basis für Kapitalinvestitionen in Prozessautomatisierung und -verbesserung
- notwendige Aktionen:
  - ⇨ kontinuierlicher Schwerpunkt auf Prozessvermessung und -verbesserung (zur Fehlervermeidung)



Stand von Organisationen im Jahre 2000 (2007):

Daten vom Software Engineering Institute (SEI) aus dem Jahr 2000 (2007) unter **http://www.sei.cmu.edu/appraisal-program/profile/profile.html** (Die Daten in Klammern betreffen das Jahr 2007 für den Vergleich mit dem Stand im Jahr 2000, der unter obiger URL nicht mehr dokumentiert ist):

- 32,3 % (1,7 %) der Organisationen im Zustand „initial“
- 39,3 % (32,7 %) der Organisationen im Zustand „wiederholbar“
- 19,4 % (36,1 %) der Organisationen im Zustand „definiert“
- 5,4 % (4,2 %) der Organisationen im Zustand „kontrolliert“
- 3,7 % (16,4 %) der Organisationen im Zustand „optimierend“

Genauere Einführung in CMM findet man in [Dy02].



ISO 9000 und CMM im Vergleich:

- Schwerpunkt der **ISO 9001** Zertifizierung liegt auf Nachweis eines Qualitäts-managementsystems im Sinne der Norm
  - ⇨ allgemein für Produktionsabläufe geeignet
  - ⇨ genau ein Reifegrad wird zertifiziert
- **CMM** konzentriert sich auf Qualitäts- und Produktivitätssteigerung des gesamten Softwareentwicklungsprozesses
  - ⇨ auf Softwareentwicklung zugeschnitten
  - ⇨ dynamisches Modell mit kontinuierlichem Verbesserungsdruck
- ISO-Norm **SPiCE** (http://www.sqi.gu.edu.au/SPICE/) integriert und vereinheit-licht CMM und ISO 9000 (als ISO/IEC 15504)



SPiCE = Software Process Improvement and Capability dEtermination:

Internationale Norm für **Prozessbewertung** (und Verbesserung). Sie bildet einheitlichen Rahmen für Bewertung der Leistungsfähigkeit von Organisationen, deren Aufgabe Entwicklung oder Erwerb, Lieferung, Einführung und Betreuung von Software-Systemen ist. Norm legt Evaluierungsprozess und Darstellung der Evaluierungsergebnisse fest.

Unterschiede zu CMM:

- orthogonale Betrachtung von Reifegraden und Aktivitätsbereichen
- deshalb andere Definition der 5 Reifegrade (z.B. „1“ = alle Aktivitäten eines Bereiches sind vorhanden, Qualität der Aktivitäten noch unerheblich, ...)
- jedem Aktivitätsbereich oder Unterbereich kann ein anderer Reifegrad zugeordnet werden



Aktivitätsbereiche von SPiCE:

- **Customer-Supplier-Bereich**:
  - Aquisition eines Projektes (Angebotserstellung, ...)
  - ...
- **Engineering-Bereich**:
  - Software-Entwicklung (Anforderungsanalyse, ... , Systemintegration)
  - Software-Wartung
- **Support-Bereich**:
  - Qualitätssicherung
  - ...
- **Management-Bereich**:
  - Projekt-Management
  - ...
- **Organisations-Bereich**:
  - Prozess-Verbesserung
  - ...



CMMI = Capability Maturity Model Integration (neue Version von CMM):

CMMI ist die **neue Version des Software Capability Maturity Model**. Es ersetzt nicht nur verschiedene Qualitäts-Modelle für unterschiedliche Entwicklungs-Disziplinen (z.B. für Software-Entwicklung oder System-Entwicklung), sondern integriert diese in einem neuen, modularen Modell. Dieses modulare Konzept ermöglicht zum einen die Integration weiterer Entwicklungs-Disziplinen (z.B. Hardware-Entwicklung), und zum anderen auch die Anwendung des Qualitätsmodells in übergreifenden Disziplinen (z.B. Entwicklung von Chips mit Software).

Geschichte von CMM und CMMI:

- ⇨ 1991 wird Capability Maturity Model 1.0 herausgegeben
- ⇨ 1993 wird CMM überarbeitet und in der Version 1.1 bereitgestellt
- ⇨ 1997 wird CMM 2.0 kurz vor Verabschiedung vom DoD zurückgezogen
- ⇨ 2000 wird CMMI als Pilotversion 1.0 herausgegeben
- ⇨ 2002 wird CMMI freigegeben
- ⇨ Ende 2003 ist die Unterstützung CMM ausgelaufen



Eigenschaften von CMMI:

Es gibt **Fähigkeitsgrade** für einzelne Prozessgebiete (ähnlich zu SPiCE):

0 - Incomplete:
Ausgangszustand, keine Anforderungen

1 - Performed:
die spezifischen Ziele des Prozessgebiets werden erreicht

2 - Managed:
der Prozess wird gemanagt

3 - Defined:
der Prozess wird auf Basis eines angepassten Standard-Prozesses gemanagt und verbessert

4 - Quantitatively Managed:
der Prozess steht unter statistischer Prozesskontrolle

5 - Optimizing:
der Prozess wird mit Daten aus der statistischen Prozesskontrolle verbessert



Eigenschaften von CMMI - Fortsetzung:

Es gibt **Reifegrade**, die Fähigkeitsgrade auf bestimmten Prozessgebieten erfordern (ähnlich zu CMM):

1- Initial:
keine Anforderungen, diesen Reifegrad hat jede Organisation automatisch

2 - Managed:
die Projekte werden gemanagt durchgeführt und ein ähnliches Projekt kann erfolgreich wiederholt werden

3 - Defined:
die Projekte werden nach einem angepassten Standard-Prozess durchgeführt, und es gibt eine kontinuierliche Prozessverbesserung

4 - Quantitatively Managed:
es wird eine statistische Prozesskontrolle durchgeführt

5 - Optimizing:
die Prozesse werden mit Daten aus statistischen Prozesskontrolle verbessert



Konsequenzen für die „eigene“ Software-Entwicklung:

Im Rahmen von Studienarbeiten, Diplomarbeiten, … können Sie keinen CMM-Level-5-Software-Entwicklungsprozess verwenden, aber:

- Einsatz von Werkzeugen für **Anforderungsanalyse, Modellierung** und **Projektplanung**
  - ⇨ in dieser Vorlesung behandelt
- Einsatz von **Konfigurations- und Versionsmanagement**-Software
  - ⇨ wird in „Software Engineering II“ behandelt
- Einsatz von Werkzeugen für systematisches **Testen, Messen** der Produktqualität
  - ⇨ wird in „Software Engineering II“ behandelt
- Einsatz von „**Extreme Programming**“-Techniken
  - ⇨ siehe „Software-Praktikum“ und z.B. [Be99] vom Erfinder Kent Beck
- Einsatz von Techniken zur Verbesserung des „persönlichen“ Vorgehensmodells
  - ⇨ siehe [Hu96] über den „**Personal Software Process**“ (Buchautor Humphrey ist einer der „Erfinder“ von CMM)



10.6 Projektpläne und Projektorganisation

Am Ende der Machbarkeitsstudie steht die Erstellung eines Projektplans mit

- ⇨ Identifikation der einzelnen **Arbeitspakete**
- ⇨ **Terminplanung** (zeitliche Aufeinanderfolge der Pakete)
- ⇨ **Ressourcenplanung** (Zuordnung von Personen zu Paketen, …)

Hier wird am deutlichsten, dass eine Machbarkeitsstudie ohne ein grobes Design der zu erstellenden Software nicht durchführbar ist, da:

- ⇨ Arbeitspakete ergeben sich aus der Struktur der Software
- ⇨ Abhängigkeiten und Umfang der Pakete ebenso
- ⇨ Realisierungsart der Pakete bestimmt benötigte Ressourcen

Konsequenz: Projektplanung und -organisation ist ein fortlaufender Prozess.
Zu Projektbeginn hat man nur einen groben Plan, der sukzessive verfeinert wird.



Terminologie:

- **Prozessarchitektur** = grundsätzliche Vorgehensweise einer Firma für die Beschreibung von Software-Entwicklungsprozessen (Notation, Werkzeuge)
- **Prozessmodell** = Vorgehensmodell = von einer Firma gewählter Entwicklungsprozess (Wasserfallmodell oder RUP oder …)
- **Projektplan** = an einem Prozessmodell sich orientierender Plan für die Durchführung eines konkreten Projektes
- **Vorgang** = Aufgabe = Arbeitspaket = abgeschlossene Aktivität in Projektplan, die
  - ⇨ bestimmte Eingaben (Vorbedingungen) benötigt und Ausgaben produziert
  - ⇨ Personal und (sonstige) Betriebsmittel für Ausführung braucht
  - ⇨ eine bestimmte Zeitdauer in Anspruch nimmt
  - ⇨ und Kosten verursacht und/oder Einnahmen bringt
- **Phase** = Zusammenfassung mehrerer zusammengehöriger Vorgänge zu einem globalen Arbeitsschritt
- **Meilenstein** = Ende einer Gruppe von Vorgängen (Phase) mit besonderer Bedeutung (für die Projektüberwachung) und wohldefinierten *Ergebnissen*



Annahmen für MVRS-Projektplanung über (Grob-)Design der Software:

- es wird eine (sehr einfache) Dreischichtenarchitektur verwendet
- alle Daten werden in einer Datenbank gespeichert
  - ⇨ Datenbankschema muss entworfen werden
  - ⇨ Anfrageoperationen setzen auf Schema auf und geben Ergebnisse in Listenform aus
  - ⇨ Update-Operationen setzen auf Datenbankschema auf und sind unabhängig von Anfrageoperationen
- die Realisierung der Benutzeroberfläche ist von der Datenbank entkoppelt, für den sinnvollen Modultest der Operationen braucht man aber die Oberfläche
- um das Beispiel nicht zu kompliziert zu gestalten, wird das Wasserfallmodell mit Integration der einzelnen Teilsysteme im „Big Bang"-Testverfahren verwendet
- gedruckte Manuale und Online-Hilfe enthalten Screendumps der Benutzer-oberfläche (teilweise parallele Bearbeitung trotzdem möglich)



Aufteilung des MVRS-Projekts in Arbeitspakete (Aufgaben):

Im obigen Bild fehlen noch die Angaben für die geschätzten Arbeitszeiten zur Bearbeitung der einzelnen Teilaufgaben des „Motor Vehicle Reservation System“-Projekts



Planung von Arbeitspaketabhängigkeiten (mit PERT-Charts) - Idee:

PERT-Chart = **P**roject **E**valuation and **R**eview **T**echnique



Planung von Arbeitspaketabhängigkeiten - früheste Anfangszeiten:

A:1. Tag
Analyse
10 Tage

A:11. Tag
Design
5 Tage

A:16. Tag
Listen-
aufbau
6 Tage

A:16. Tag
Datenbank-
schema
10 Tage

A:16. Tag
Benutzer-
oberfläche
18 Tage

A:34. Tag
Anfrage-
operationen
10 Tage

A:34. Tag
Update-
operationen
10 Tage

A:44. Tag
Integration
(Systemtest)
10 Tage

A:34. Tag
Online-Hilfe
15 Tage

A:34. Tag
Manual
20 Tage

A:54. Tag
Abnahme
10 Tage

Vorwärtsberechnung von Startdatum für Aufgabe:

früheste Anfangszeit = Max(früheste Vorgängeranfangszeit + Vorgängerdauer)



Planung von Arbeitspaketabhängigkeiten - späteste Endzeiten:

A:1. / E:10.
Analyse
10 Tage

A:11. / E:15.
Design
5 Tage

A:16. / E:33.
Listen-
aufbau
6 Tage

A:16. / E:33.
Datenbank-
schema
10 Tage

A:16. / E:33.
Benutzer-
oberfläche
18 Tage

A:34. / E:43.
Anfrage-
operationen
10 Tage

A:34. / E:43.
Update-
operationen
10 Tage

A:44. / E:53.
Integration
(Systemtest)
10 Tage

A:34. / E:53.
Online-Hilfe
15 Tage

A:34. / E:53.
Manual
20 Tage

A:54. / E:63.
Abnahme
10 Tage

Rückwärtsberechnung von Enddatum für Aufgabe:

späteste Endzeit = Min(späteste Nachfolgerendzeit - Nachfolgerdauer)



Planung von Arbeitspaketabhängigkeiten - kritische Pfade u. Aufgaben:

Berechnung kritischer Pfade mit kritischen Aufgaben:

für **kritische Aufgabe** gilt: früheste Anfangszeit + Dauer = späteste Endzeit + 1



Zusammenfassung der Berechnung:

- **Vorwärtsberechnung** frühester Anfangszeiten für Aufgaben:
  - ⇨ früheste **A**nfangszeit erster Aufgabe = 1
  - ⇨ früheste **A**nfangszeit von Folgeaufgabe = Maximum(früheste Anfangszeit + Dauer einer Vorgängeraufgabe)
- **Rückwärtsberechnung** spätester Endzeiten für Aufgaben:
  - ⇨ späteste **E**ndzeit letzter Aufgabe = früheste Anfangszeit + Dauer - 1
  - ⇨ späteste **E**ndzeit von Vorgängeraufgabe = Minimum(späteste Endzeit - Dauer einer Nachfolgeraufgabe)
- **kritische Aufgabe** auf kritischem Pfad (Anfangs- und Endzeiten liegen fest):
  - ⇨ früheste Anfangszeit = späteste Anfangszeit := späteste Endzeit - Dauer +1
  - ⇨ späteste Endzeit = früheste Endzeit := früheste Anfangszeit + Dauer - 1
- **kritische Kante** auf kritischem Pfad (zwischen zwei kritischen Aufgaben):
  - ⇨ früheste Endzeit Kantenquelle + 1 = späteste Anfangszeit Kantensenke
- **Pufferzeit** nichtkritischer Aufgabe (Zeit um die Beginn verschoben werden kann):
  - ⇨ Pufferzeit = späteste Endzeit - früheste Endzeit



Probleme mit der Planung des Beispiels:

- zu viele Aufgaben liegen auf kritischen Pfaden (wenn kritische Aufgabe länger als geschätzt dauert, schlägt das auf Gesamtprojektlaufzeit durch)
  - ⇨ an einigen Stellen zusätzliche Pufferzeiten einplanen (um Krankheiten Fehlschätzungen, … auffangen zu können)
- einige eigentlich parallel ausführbare Aufgaben sollen von der selben Person bearbeitet werden
  - ⇨ Ressourcenplanung für Aufgaben durchführen
  - ⇨ Aufgaben serialisieren um „Ressourcenkonflikte“ aufzulösen
  - ⇨ (oder: weitere Personen im Projekt beschäftigen)
- Umrechnung auf konkrete Datumsangaben fehlt noch
  - ⇨ Berücksichtigung von Wochenenden (5 Arbeitstage pro Woche)
  - ⇨ ggf. auch Berücksichtigung von Urlaubszeiten



Verfeinerung von Abhängigkeiten und Zeitplanung:

Bislang wird bei allen abhängigen Aktivitäten vorausgesetzt:

- ⇨ **„Finish-to-Start"-Folge (FS) = Normalfolge**:
  Nachfolgeaktivität kann erst bearbeitet werden, wenn Vorgängeraktivität vollständig abgeschlossen ist

Manchmal lassen sich aber abhängige Aktivitäten auch parallel bearbeiten:

- ⇨ **„Start-to-Start"-Folge (SS) = Anfangsfolge**:
  Aktivitäten müssen gleichzeitig beginnen
- ⇨ **„Finish-to-Finish"-Folge (FF) = Endfolge**:
  Aktivitäten müssen gleichzeitig enden
- ⇨ **„Start-to-Finish"-Folge (SF) = Sprungfolge**:
  Umkehrung der Nachfolgebeziehung (relativ sinnlose Beziehung)

Des weiteren sind manchmal zusätzliche Verzögerungszeiten sinnvoll:

- ⇨ frühestmöglicher Beginn einer Aktivität wird um **Verzögerungszeit (lag)** nach vorne oder hinten verschoben (für feste Puffer, Überlappungen)



Balkendiagramm (Gantt Chart, 1917 von Henry Gantt erfunden) - Idee:

14.4.03 28.4.03 5.5.03 19.5.03 2.6.03 16.6.03 30.6.03 11.7.03

1. Analyse
2. Design
3.1 DB-Schema
3.2 DB-Updates
3.3 DB-Queries
3.4 Listen
3.5 Benutzeroberfläche
5.1 Manual
5.2 Online-Hilfe
4. Integration (Systemtest)
6. Abnahme

Zeitraum für Paketbearbeitung: Nettozeit: Pufferzeit (float):



Balkendiagramm mit Personalplanung:

14.4.03 | 28.4.03 | 5.5.03 | 19.5.03 | 2.6.03 | 16.6.03 | 30.6.03 | 11.7.03

Valerie (verantwort-lich für alle Phasen): Analyse | Design | DB-Schema | Listen | DB-Updates | Integration | Abnahme

Edeltraud: Analyse | Manual | Online-Hilfe | Abnahme

Otto: **Urlaub** | Benutzeroberfläche | DB-Queries | Integration | Abnahme

Nettozeit:

Pufferzeit: (float)

Achtung:

Durch Ressourcenzuteilung entstehen zusätzliche Zeit-restriktionen (z.B. Listen nach DB-Schema)



10.7 Aufwands- und Kostenschätzung

Die **Kosten** eines Softwareproduktes und die **Entwicklungsdauer** werden im wesentlichen durch den personellen Aufwand bestimmt. Bislang haben wir vorausgesetzt, dass der personelle Aufwand bekannt ist, hier werden wir uns mit seiner Berechnung bzw. Schätzung befassen.

Der **personelle Aufwand** für die Erstellung eines Softwareproduktes ergibt sich aus

- ⇨ dem „**Umfang**" des zu erstellenden Softwareprodukts
- ⇨ der geforderten **Qualität** für das Produkt

Übliches Maß für Personalaufwand:

Mitarbeitermonate (MM) oder Mitarbeiterjahre (MJ):

1 MJ ≈ 10 MM (wegen Urlaub, Krankheit, …)

Übliches Maß für Produktumfang:

„Lines of Code" (LOC) = Anzahl Zeilen der Implementierung ohne Kommentare



Schätzverfahren im Überblick:

- **Analogiemethode**: Experte vergleicht neues Projekt mit bereits abgeschlossenen ähnlichen Projekten und schätzt Kosten „gefühlsmäßig“ ab
  - ⇨ Expertenwissen lässt sich schwer vermitteln und nachvollziehen
- **Prozentsatzmethode**: aus abgeschlossenen Projekten wird Aufwandsverteilung auf Phasen ermittelt; anhand beendeter Phasen wird Projektrestlaufzeit geschätzt
  - ⇨ funktioniert allenfalls nach Abschluss der Analysephase
- **Parkinsons Gesetz**: die Arbeit ist beendet, wenn alle Vorräte aufgebraucht sind
  - ⇨ praxisnah, realistisch und wenig hilfreich …
- **Price to Win**: die Software-Kosten werden auf das Budget des Kundens geschätzt
  - ⇨ andere Formulierung von „Parkinsons Gesetz“, führt in den Ruin …
- **Gewichtungsmethode**: Bestimmung vieler Faktoren (Erfahrung der Mitarbeiter, verwendete Sprachen, …) und Verknüpfung durch mathematische Formel
  - ⇨ LOC-basierter Vertreter: COnstructive COst MOdel (COCOMO)
  - ⇨ FP-basierte Vertreter: Function-Point-Methoden in vielen Varianten



Softwareumfang = Lines of Code?

Die „**Lines of Code**" als Ausgangsbasis für die Projektplanung (und damit auch zur Überwachung der Produktivität von Mitarbeitern) zu verwenden ist **fragwürdig**, da:

- ⇨ Codeumfang erst mit Abschluss der Implementierungsphase bekannt ist
- ⇨ selbst Architektur auf Teilsystemebene noch unbekannt ist
- ⇨ Wiederverwendung mit geringeren LOC-Zahlen bestraft wird
- ⇨ gründliche Analyse, Design, Testen, … zu geringerer Produktivität führt
- ⇨ Anzahl von Codezeilen abhängig vom persönlichen Programmierstil ist
- ⇨ Handbücher schreiben, … ungenügend berücksichtigt wird

Achtung:

Die starke **Abhängigkeit** der LOC-Zahlen **von** einer **Programmiersprache** ist zulässig, da Programmiersprachenwahl (großen) Einfluss auf Produktivität hat.



Einfluss von Programmiersprache auf Produktivität:

	Analyse	Design	Codierung	Test	Sonstiges
C	3 Wochen	5 Wochen	8 Wochen	10 Wochen	2 Wochen
Smalltalk	3 Wochen	5 Wochen	2 Wochen	6 Wochen	2 Wochen

Konsequenzen für die Gesamtproduktivität:

	Programmgröße	Aufwand	Produktivität
C	2.000 LOC	28 Wochen	70 LOC/Woche
Smalltalk	500 LOC	18 Wochen	27 LOC/Woche

Fazit:

- ☞ Produktivität kann **nicht** in „Lines Of Code pro Zeiteinheit“ sinnvoll gemessen werden (sonst wäre Programmieren in Assembler die beste Lösung)
- ☞ also: Vorsicht mit Einsatz von Maßzahlen (keine sozialistische Planwirtschaft)



Softwareumfang = Function Points!

Bei der **Function-Point-Methode** zur Kostenschätzung wird der Softwareumfang anhand der Produktanforderungen aus dem Lastenheft geschätzt. Es gibt inzwischen einige Spielarten; hier wird (weitgehend) der Ansatz der **International Function Point Users Group (IFPUG)** vorgestellt, siehe auch http://www.ifpug.org.

Jede Anforderung wird gemäß IFPUG einer von 5 Kategorien zugeordnet [ACF97]:

1. **Eingabedaten** (über Tastatur, CD, externe Schnittstellen, …)
2. **Ausgabedaten** (auf Bildschirm, Papier, externe Schnittstelle, …)
3. **Abfragen** (z.B. SQL-Queries auf einem internen Datenbestand)
4. **Datenbestände** (sich ändernde interne Datenbankinhalte)
5. **Referenzdateien** (im wesentlichen unveränderliche Daten)

Dann werden **Function-Points (FPs)** berechnet, bewertet, … .



Datenbestände = Internal Logical File (ILF) = Interne Entitäten:

Unter einer **internen (Geschäfts-)Entität** definiert die IFPUG eine aus Anwendersicht logisch zusammengehörige Gruppe vom Softwaresystem verwalteter Daten, also z.B.:

- ⇨ eine Gruppe von Produktdaten des Lastenheftes in der Machbarkeitsstudie
- ⇨ Klassen mit Attributen u. Beziehungen eines Paketes aus Modellen im Pflichtenheft in der Analysephase

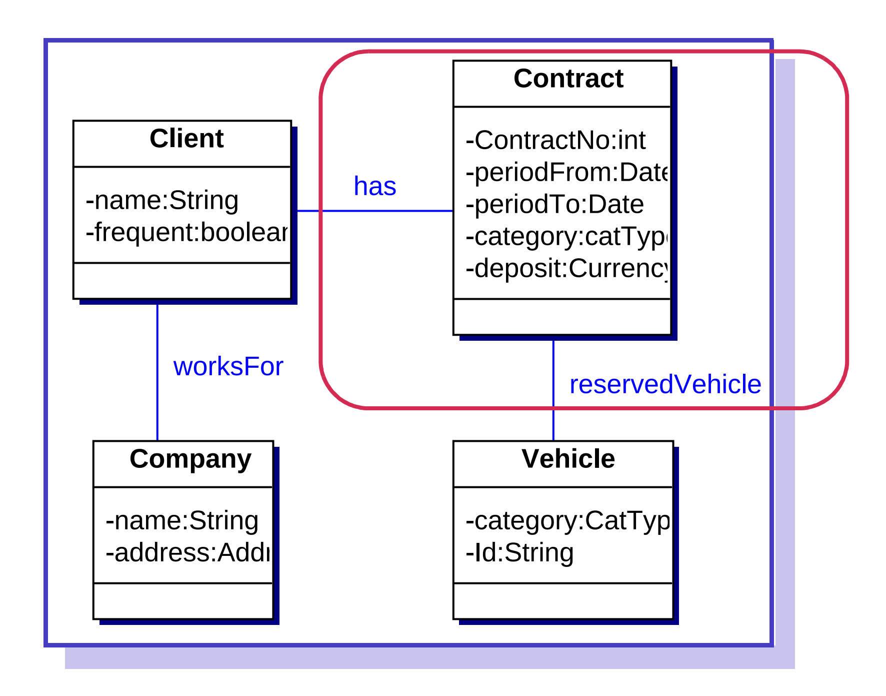
Es werden Datenelementtypen (Attribute) sowie Entitätstypen (Klassen, Sätze) und zusätzlich Beziehungstypen (Assoziationen) gezählt. Anhand dieser Zählung wird Komplexität eines Datenbestandes wie folgt bestimmt:

einfach = 7 FPs, mittel = 10 FPs oder komplex = 15 FPs

Interne Entitäten	Anzahl Attribute <= 19	19 < Anzahl Attribute ≤ 50	Anzahl Attribute > 50
Klassen+Assoz. <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
2 ≤ Klassen+Assoz. ≤ 5	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Klassen+Assoz. > 5	mittlere Komplexität	hohe Komplexität	hohe Komplexität



Beispiel für Bewertung eines internen Datenbestandes:



Datenbasis mit Verträgen

Die Datenbasis besteht aus



Referenzdateien = External Interface File = (EIF) = Externe Entitäten:

Unter einer **externen (Geschäfts-)Entität** definiert die IFPUG eine aus Anwender-sicht logisch zusammengehörige Gruppe vom System benutzter aber nicht selbst ver-walteter Daten.

Wieder werden Datenelementtypen (Attribute) sowie Entitätstypen (Klassen, Sätze) und zusätzlich Beziehungstypen (Assoziationen) gezählt. Anhand dieser Zählung wird Komplexität eines Datenbestandes wie folgt bestimmt:

einfach = 5 FPs, mittel = 7 FPs oder komplex = 10 FPs

Externe Entitäten	Anzahl Attribute <= 19	19 < Anzahl Attribute ≤ 50	Anzahl Attribute > 50
Klassen+Assoz. <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
2 ≤ Klassen+Assoz. ≤ 5	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Klassen+Assoz. > 5	mittlere Komplexität	hohe Komplexität	hohe Komplexität

Es werden weniger FPs als bei internen Entitäten vergeben, da die betrachteten Daten-bestände nur eingelesen aber nicht verwaltet werden müssen.



(Externe) Eingabedaten = External Input (EI):

Eingabedaten für **Elementarprozess**, der Daten oder Steuerinformationen des Anwenders verarbeitet, aber keine Ausgabedaten liefert. Es handelt sich dabei um den kleinsten selbständigen Arbeitsschritt in der Arbeitsfolge eines Anwenders, als etwa:

- ⇨ Produktfunktionen des Lastenheftes in der Machbarkeitsstudie
- ⇨ „Use Cases“ aus Pflichtenheft in der Analysephase

Gezählt werden für jeden Elementarprozess die Anzahl seiner als Eingabe verwende-ten Entitätstypen (Klassen, Sätze) und deren Datenelementtypen (Attribute, Felder). Anhand dieser Zählung wird Komplexität des Elementarprozesses wie folgt bestimmt:

einfach = 3 FPs, mittel = 4 FPs oder komplex = 6 FPs

Externe Eingabe	Anzahl Attribute <= 4	4 < Anzahl Attribute ≤ 15	Anzahl Attribute > 15
Anzahl Klassen <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
Anzahl Klassen = 2	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Anzahl Klassen > 2	mittlere Komplexität	hohe Komplexität	hohe Komplexität



Beispiel für die Bewertung externer Eingabedaten:

- Reservierungsvertrag **neu erstellen** mit Beginn- und Endedatum, gewünschter Fahrzeugkategorie, Kaution, Kunde und Fahrzeug (eine eindeutige Vertragsnummer wird automatisch angelegt)
- Reservierungsvertrag mit Vertragsnummer **verändern** (alle Werte ausser Kunde)
- Reservierungsvertrag mit Vertragsnummer **löschen**
- Daten zu einem Reservierungsvertrag mit Vertragsnummer **anzeigen**

Es wird wie folgt gezählt:



Externe Ausgaben = External Output (EO):

Ausgabedaten eines **Elementarprozesses** (Produktfunktion, Use Case), der Anwender Daten oder Steuerinformationen liefert. Achtung: der Elementarprozess darf keine Eingabedaten benötigen; ansonsten handelt es sich um eine „Externe Abfrage“ oder ... .

Gezählt werden für jeden Elementarprozess die Anzahl seiner als Ausgabe verwendeten Entitätstypen (Klassen, Sätze) und deren Datenelementtypen (Attribute, Felder). Anhand dieser Zählung wird Komplexität des Elementarprozesses wie folgt bestimmt:

einfach = 4 FPs, mittel = 5 FPs oder komplex = 7 FPs

Externe Ausgaben	Anzahl Attribute <= 5	5 < Anzahl Attribute ≤ 19	Anzahl Attribute > 19
Anzahl Klassen <= 1	einfache Komplexität	einfache Komplexität	mittlere Komplexität
2 ≤ Anzahl Klassen ≤ 3	einfache Komplexität	mittlere Komplexität	hohe Komplexität
Anzahl Klassen > 3	mittlere Komplexität	hohe Komplexität	hohe Komplexität



Beispiel für die Bewertung externer Ausgabedaten:

- Daten zu einem Reservierungsvertrag mit Vertragsnummer **anzeigen** (mit Name des Kunden und tatsächlicher Fahrzeugkategorie zusätzlich zu Kundennummer und Fahrzeugnummer)
- **alle Reservierungsverträge** zu einem Kunden mit Kundennummer anzeigen
- die **Kosten** für alle Reservierungsverträge eines Kunden anzeigen

Es wird wie folgt gezählt:



Externe Abfragen = External Inquiry (EQ):

Betrachtet werden **Elementarprozesse** (Produktfunktion, Use Case), die anhand von Eingaben Daten des internen Datenbestandes ausgeben (ohne auf diesen Daten komplexe Berechnungen durchzuführen).

Nach den Regeln für „Externe Eingaben“ werden die Eingabedaten bewertet, nach den Regeln für „Externe Ausgaben“ die Ausgabedaten; anschließend wird die höhere Komplexität übernommen und wie folgt umgerechnet:

einfach = 3 FPs, mittel = 4 FPs oder komplex = 6 FPs

Achtung: ein Elementarprozess, der Eingabedaten zur Suche nach intern gespeicherten Daten benötigt und vor der Ausgabe **komplexe Berechnungen** durchführt, wird anders behandelt. In diesem Fall wird nicht das Maximum gebildet, sondern die **Summe** der FPs von „Externe Eingabe“ und „Externe Ausgabe“.



Beispiel für die Bewertung einer externen Abfrage:

- **alle Reservierungsverträge** zu einem Kunden mit Kundennummer anzeigen (mit Kundename und -nummer sowie Fahrzeugkategorie und -nummer)
- die **Kosten** für alle Reservierungsverträge eines Kunden anzeigen, der über seine Kundennummer festgelegt wird.

Es wird wie folgt gezählt:



Überblick über die FP-Methode - 1 [Ba98]:

1. Schritt:
Kategorisierung für jede Anforderung

2. Schritt:
Klassifizierung jeder Anforderung

3. Schritt: Eintrag der jeweiligen Anzahl in das Berechnungs-formular und Ermittlung der unbewerteten FPs

4. Schritt:
Bewertung der Einflußfaktoren



Überblick über die FP-Methode - 2:

5. Schritt: Berechnung der bewerteten FPs

6. Schritt: Ablesen des Aufwandes

7. Schritt: Aktualisierung der Wertepaare, Neu-berechnung der Aufwandskurve



Berechnungsformel für die FP-Methode:

Berechnungsformel für die FP-Methode:

Kategorie	Anzahl	Klassifizierung	Gewichtung	Zeilensumme
Eingabedaten		einfach	x 3	=
		mittel	x 4	=
		komplex	x 6	=
Abfragen		einfach	x 3	=
		mittel	x 4	=
		komplex	x 6	=
Ausgaben		einfach	x 4	=
		mittel	x 5	=
		komplex	x 7	=
Datenbestände		einfach	x 7	=
		mittel	x 10	=
		komplex	x 15	=
Referenzdaten		einfach	x 5	=
		mittel	x 7	=
		komplex	x 10	=
Summe			E1	=
Einflußfaktoren (ändern den *Function Point*-Wert um ± 30%)		1 Verflechtung mit anderen Anwendungssystemen (0–5)		=
		2 Dezentrale Daten, dezentrale Verarbeitung (0–5)		=
		3 Transaktionsrate (0–5)		=
		4 Verarbeitungslogik		
		a Rechenoperationen (0–10)		=
		b Kontrollverfahren (0–5)		=
		c Ausnahmeregelungen (0–10)		=
		d Logik (0–5)		=
		5 Wiederverwendbarkeit (0–5)		=
		6 Datenbestands-Konvertierungen (0–5)		=
		7 Anpaßbarkeit (0–5)		=
Summe der 7 Einflüsse		E2		=
Faktor Einflußbewertung = E2 / 100 + 0,7		E3		=
Bewertete *Function Points:* E1 * E3				=



Zusätzliche Einflussfaktoren:

Die vorige Tabelle unterscheidet sieben Einflussfaktoren; andere Quellen nennen 14 bzw. **19 verschiedene Faktoren**, die Werte von 0 bis 5 erhalten (siehe [Hu99]):

1. Komplexität der Datenkommunikation
2. Grad der verteilten Datenverarbeitung
3. geforderte Leistungsfähigkeit
4. Komplexität der Konfiguration (Zielplattform)
5. Anforderung an Transaktionsrate
6. Prozentsatz interaktiver Dateneingaben
7. geforderte Benutzerfreundlichkeit
8. interaktive bzw. Online-Pflege des internen Datenbestandes
9. Komplexität der Verarbeitungslogik



Zusätzliche Einflussfaktoren - Fortsetzung:

10. geforderter Grad der Wiederverwendbarkeit
11. benötigte Installationshilfen
12. leichte Bedienbarkeit (Grad der Automatisierung der Bedienung)
13. Mehrfachinstallationen (auf verschiedenen Zielplattformen)
14. Grad der gefoderten Änderungsfreundlichkeit
15. Randbedingungen anderer Anwendungen
16. Sicherheit, Datenschutz, Prüfbarkeit
17. Anwenderschulung
18. Datenaustausch mit anderen Anwendungen
19. Dokumentation



Ermittlung der FP-Aufwandskurve:

- beim ersten Projekt muss man auf **bekannte Kurven** (für ähnliche Projekte) zurückgreifen (IBM-Kurve mit FP = 26 * $MM^{0,8}$, VW AG Kurve, …)
- alternativ kann man eigene abgeschlossene Projekte **nachkalkulieren**, allerdings:
  - ⇨ Nachkalkulationen sind aufwändig
  - ⇨ Dokumentation von Altprojekten oft unvollständig
  - ⇨ oft gibt es nur noch den Quellcode (keine Lasten- oder Pflichtenhefte)
  - ⇨ Kosten (Personenmonate) alter Projekte oft unklar (wurden Überstunden berücksichtigt, welche Aktivitäten wurden mitgezählt, ...)
- das Verhältnis von MM zu FP bei abgeschlossenen eigenen Projekten wird zur nachträglichen „**Kalibrierung**“ der Kurve benutzt:
  - ⇨ neues Wertepaar wird hinzugefügt oder neues Wertepaar ersetzt ältestes Wertepaar
  - ⇨ Frage: was für eine Funktion benutzt man für Interpolation von Zwischenwerten (meist nicht linear, sondern eher quadratisch oder gar exponentiell)



Nachkalkulation von Projekten mit „Backfiring“-Methode:

Bei alten Projekten gibt es oft nur noch den Quellcode und keine Lasten- oder Pflichtenhefte, aus denen FPs errechnet werden können. In solchen Fällen versucht man FPs aus Quellcode wie folgt nach [Jo92] rückzurechnen:

Sprache	**Lines of Code / Function Points**		
Komplexität	**niedrig**	**mittel**	**hoch**
Assembler	200	320	450
C	60	128	170
Fortran	75	107	160
COBOL	65	107	150
C++	30	53	125
Smalltalk	15	21	40

Achtung:

LOC in Programmiersprache X pro MM Codierung angeblich nahezu konstant

$\Rightarrow$ damit ist z.B. Produktivität beim Codieren in Smalltalk 4 mal höher als in C



Vorgehensweise bei Kostenschätzung (mit FP-Methode):

1. Festlegung des verwendeten **Ansatzes**, Bereitstellung von Unterlagen
2. **Systemgrenzen** festlegen (was gehört zum Softwaresystem dazu)
3. **Umfang** des zu erstellenden Softwaresystems (in FPs) **messen**
4. Umrechnung von Umfang (FPs) in **Aufwand** (MM) mit Aufwandskurve
5. **Zuschläge** einplanen (für unvorhergesehene Dinge, Schätzungenauigkeit)
6. Aufwand auf Phasen bzw. Iterationen **verteilen**
7. Umsetzung in **Projektplan** mit Festlegung von **Teamgröße**
8. Aufwandschätzung **prüfen** und dokumentieren
9. Aufwandschätzung für Projekt während Laufzeit regelmäßig **aktualisieren**
10. Datenbasis für eingesetztes Schätzverfahren aktualisieren, Verfahren **verbessern**



Einplanung von Zuschlägen (Faustformel nach Augustin):

Korrekturfaktor **$K = 1,8 / (1 + 0,8\ F^3)$**

für Zuschläge mit *F* als geschätzter Fertigstellungsgrad der Software.

Problem mit der Berechnung des Fertigstellungsgrades der Software:

Der Wert *F* ist vor Projektende unbekannt, muss also selbst geschätzt werden als

F = bisheriger Aufwand / (bisheriger Aufwand + geschätzter Restaufwand)

Modifizierte Formel für korrigierte Aufwandsschätzung:

$$MM_g = MM_e + MM_k = MM_e + MM_r * 1,8 / (1 + 0,8\ (MM_e / (MM_e + MM_r))^3\ )$$

MM_g = korrigierter geschätzter Gesamtaufwand in Mitarbeitermonaten
MM_e = bisher erbrachter Aufwand in Mitarbeitermonaten
MM_k = korrigierter geschätzter Restaufwand in Mitarbeitermonaten
MM_r = geschätzter Restaufwand in Mitarbeitermonaten



Erläuterungen zu Korrekturfaktor für Kostenschätzung:

- zu **Projektbeginn** ist *F = 0*, da noch kein Aufwand erbracht wurde; damit wird der geschätzte Aufwand um 80% nach oben korrigiert
- am **Projektende** ist *F = 1*, da spätestens dann die aktuelle Schätzung mit tatsächlichem Wert übereinstimmen sollte, es gibt also keinen Aufschlag mehr
- Unsicherheiten in der Schätzung nehmen nicht **nicht linear** ab, da Wissenszuwachs über zu realisierende Softwarefunktionalität und technische Schwierigkeiten im Projektverlauf keinesfalls linear ist
- im Laufe des Projektes wird Fertigstellungsgrad *F* nicht immer zunehmen, sondern ggf. auch abnehmen, wenn Schätzungen sich als **zu optimistisch** erwiesen haben
- Auftraggeber wird mit Zuschlag von 80% auf geschätzte Kosten nicht zufrieden sein, deshalb werden inzwischen manchmal Verträge geschlossen, bei denen nur **Preis je realisiertem FP** vereinbart wird:
  - ⇨ Risiko für zu niedrige Schätzung von FPs liegt bei Auftraggeber
  - ⇨ Risiko für zu niedrige Umrechnung v. FPs in MM liegt bei Auftragnehmer



Aufwandsverteilung auf Phasen bzw. Entwicklungsaktivitäten:

Hat man den Gesamtaufwand für ein Softwareentwicklungsprojekt geschätzt, muss man selbst bei einer ersten Grobplanung schon die ungefähre **Länge einzelner Phasen** oder Iterationen festlegen:

- für die Aufteilung des Aufwandes auf Phasen bzw. Aktivitätsbereiche gibt es die **Prozentsatzmethode**, hier in der Hewlett-Packard-Variante aus [Ba98]:
  - ⇨ Analyseaktivitäten: 18% des Gesamtaufwandes
  - ⇨ Entwurfsaktivitäten: 19% des Gesamtaufwandes
  - ⇨ Codierungsaktivitäten: 34% des Gesamtaufwandes
  - ⇨ Testaktivitäten: 29% des Gesamtaufwandes
- für die Aufwandsberechnung **einzelner Iterationen** einer Phase wird die Zuordnung von FPs zu diesen Iterationen herangezogen oder es wird bei festgelegter Projektlänge und fester Länge von Iterationen (z.B. 4 Wochen) die Anzahl der FPs, die in einer Iteration zu behandeln sind, festgelegt



Bestimmung optimaler Entwicklungsdauer (Faustformel nach Jones):

für geringen Kommunikationsoverhead und hohen Parallelisierungsgrad:

Dauer = 2,5 * (Aufwand in MM)^{s}

s = 0,38 für Stapel-Systeme
s = 0,35 für Online-System
s = 0,32 für Echtzeit-Systeme

durchschnittliche **Teamgröße = Aufwand / Dauer**

Überlegungen zu obiger Formel:

- Anzahl der maximal sinnvoll parallel arbeitenden Mitarbeiter hängt ab von Projektart
- große Projekte dürfen nicht endlos lange laufen (also mehr Mitarbeiter)
- mit der Anzahl der Mitarbeiter wächst aber der Kommunikations- und der Verwaltungsaufwand überproportional (also weniger Mitarbeiter)
- Anzahl sinnvoll parallel zu beschäftigender Mitarbeiter während Projekt-laufzeit (Putnam-Kurve):



Rechenbeispiele für Faustformel:

Aufwand in MM	Projektart	Projektdauer	Mitarbeiterzahl
20 MM	Stapel (Batch)	7,8 Monate	2,6 Mitarbeiter
	Echtzeit (Realtime)	6,5 Monate	3,1 Mitarbeiter
200 MM	Stapel	18,7 Monate	10,7 Mitarbeiter
	Echtzeit	13,6 Monate	14,7 Mitarbeiter
2000 MM	Stapel	45,0 Monate	44,4 Mitarbeiter
	Echtzeit	28,5 Monate	70,2 Mitarbeiter

Achtung:

- ☞ geschätzter Aufwand in Mitarbeitermonaten enthält bereits organisatorischen **Overhead** für Koordination von immer mehr Mitarbeitern
- ☞ in 45,0 Monaten mit 44,4 Mitarbeitern = 2.000 MM werden also nicht 100 mal mehr FPs oder LOCs als in 7,8 Monaten mit 2,6 Mitarbeitern = 20 MM erledigt (eher nur 30 bis 40 mal mehr FPs)



Bewertung der FP-Methode:

- lange Zeit wurde LOC-basierte Vorgehensweise propagiert
- inzwischen: FP-Methode ist wohl einziges halbwegs funktionierendes Schätzverfahren
- Abweichungen trotzdem groß (insbesondere bei Einsatz „fremder" Kurven)
- Anpassung an OO-Vorgehensmodelle, moderne Benutzeroberflächen notwendig
- moderne Varianten in Form von „**Object-Point-Methode**", … sind noch nicht standardisiert und haben sich wohl noch nicht durchgesetzt
- Schätzungsfehler in der Machbarkeitsstudie sind nicht immer auf fehlerhafte Schätzmethode zurückzuführen, sondern ggf. auch auf **nicht** im Lastenheft **vereinbarte** aber **realisierte Funktionen** oder zusätzliche Umbaumaßnahmen
- bisher geschilderte Vorgehensweise **nur für Neuentwicklungen** geeignet (ohne umfangreiche Umbaumaßnahmen im Zuge iterativer Vorgehensweise)



Problematik der FP-Berechnung bei iterativer Vorgehensweise:

Bei Projekten zur **Sanierung oder Erweiterung** von Softwaresystemen bzw. bei einer stark iterativ geprägten Vorgehensweise (mit Umbaumaßnahmen) werden einem System nicht nur Funktionen hinzugefügt, sondern auch Funktionen verändert bzw. entfernt. Damit ergibt sich der Aufwand für Projektdurchführung aus:

Aufwand in MM = **Aufwand für hinzugefügte Funktionen +**
Aufwand für gelöschte Funktionen +
Aufwand für geänderte Funktionen

Vorgehensweise:

- man benötigt modifizierte Regeln für die Berechnung von FPs für **gelöschte** Funktionen (Löschen etwas einfacher als Hinzufügen, deshalb weniger FPs?)
- man benötigt modifizierte Regeln für die Berechnung von FPs für **geänderte** Funktionen (Ändern = Löschen + Hinzufügen?)



10.8 Weitere Literatur

[ACF97] V. Ambriola, R. Conradi, A. Fugetta: *Assessing Process-Centered Software Engineering Environments*, ACM TOSEM, Vol. 6, No. 3, ACM Press (1997), S. 283-328

Nicht zu alter Aufsatz mit Überblickscharakter zum Thema dieses Kapitels. Beschränkt sich allerdings im wesentlichen darauf, die drei Systeme OIKOS (Ambriola), EPOS (Conradi) und SPADE (Fugetta) der drei Autoren miteinander zu vergleichen. Es handelt sich dabei um Systeme der zweiten Generation, die in diesem Kapitel nicht vorgestellt wurden.

[Ba98] H. Balzert: *Lehrbuch der Softwaretechnik (Band 2): Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung*, Spektrum Akademischer Verlag (1998)

Hier findet man (fast) alles Wissenswerte zum Thema Management der Software-Entwicklung.

[BP84] V.R. Basili, B.T. Perricone: *Software Errors and Complexity: An Empirical Investigation*, Communications of the ACM, Vol. 27, No. 1, 42-52, ACM Press (1984)

Eine der ersten Publikationen zu empirischen Untersuchungen über den Zusammenhang von Softwarekomplexität und Fehlerhäufigkeit

[Be99] K. Beck: *Extreme Programming Explained - Embrace the Change*, Addison Wesley (1999)

Eins der Standardwerke zum Thema XP, geschrieben vom Erfinder. Mehr Bettlektüre mit Hintergrundinformationen und Motivation für XP-Techniken, als konkrete Handlungsanweisung.

[Dy02] K.M. Dymond: *CMM Handbuch*, Springer Verlag (2002)

Einfach zu verstehende schrittweise Einführung in CMM (Levels und Upgrades von einem Level zum nächsten) in deutscher Sprache.



[Hu99] R. Hürten: Function-Point-Analysis - Theorie und Praxis (Die Grundlage für ein modernes Software-Management), expert-verlag (1999), 177 Seiten

Kompaktes Buch zur Kostenschätzung mit Function-Point-Methode, das Wert auf nachvollziehbare Beispiele legt. Zusätzlich gibt es eine Floppy-Disc mit entsprechenden Excel-Sheets.

[Hu96] W.S. Humphrey: *Introduction to the Personal Software Process*, SEI Series in Software Engineering, Addison Wesley (1996)

Das CMM-Modell für den „kleinen Mann". Das Buch beschreibt wie man als einzelner Softwareentwickler von bescheidenen Anfängen ausgehend die Prinzipien der Prozessüberwachung und -verbesserung einsetzen kann. Zielsetzungen sind zuverlässigere Zeit- und Kostenschätzungen, Fehlerreduktion, … .

[JBR99] I. Jacobson, G. Booch, J. Rumbaugh: *The Unified Software Development Process*, Addison Wesley (1999)

Präsentation des auf die UML zugeschnittenen Vorgehensmodells der Softwareentwicklung, eine Variante des hier vorgestellten Rational Unified (Objectory) Prozesses.

[Jo92] C. Jones: *CASE's Missing Elements*, IEEE Spektrum, Juni 1992, S. 38-41, IEEE Computer Society Press (1992)

Enthält u.a. ein vielzitiertes Diagramm über Produktivitäts- und Qualitätsverlauf bei Einsatz von CASE.

[OHJ99] B. Oestereich (Hrsg.), P. Hruschka, N. Josuttis, H. Kocher, H. Krasemann, M. Reinhold: *Erfolgreich mit Objektorientierung: Vorgehensmodelle und Managementpraktiken für die objektorientierte Softwareentwicklung*, Oldenbourg Verlag (1999)

Ein von Praktikern geschriebenes Buch mit einer Fülle von Tipps und Tricks. Es enthält eine kurze Einführung in den "Unified Software Development Process", sowie in das V-Modell.*